JavaOS™ for Business™

JavaOS™ for Business™ Version 2.0

Device Driver Guide

June 1998

COPYRIGHT

Preface

The *JavaOS for Business Device Driver Guide* is a hands-on tutorial that describes how to write a driver for the JavaOS for Business system.

Who Should Read This Book

The main purpose of this guide is to describe how to construct a device driver. The primary audience includes driver writers and system administrators. At some point application developers may be interested in writing custom drivers to include in their applications.

What You Need To Know

This guide concentrates on the strategies and procedures for creating device drivers for the new JavaOS for Business platform. The reader should be familiar with the following topics:

- operating systems
- object-oriented programming
- Java programming
- device hardware
- bus architecture
- software development tools
- network administration

Additional Reading

Since JavaOS for Business is based on Java technology developed for the Java Development Kit (JDK), readers should be familiar with JDK 1.1.4. The following documentation provides important background information.

- *JavaOS for Business Reference Guide* is a companion document that provides an overview of JavaOS for Business architecture and its various components. The *Reference Guide* is an *under the hood* look at inter-relationships among many system components. In addition the *Network Operations Guide* that accompanies this documentation may prove a valuable source of information.
- *The Java Language Specification* (Addison-Wesley, 1996) is a rigorous definition of the Java programming language. Chapter 12, “Execution” describes bytecode execution and includes important information on the VM.
- *The Java Virtual Machine Specification* (Addison-Wesley, 1996) describes the specification of the Java Virtual Machine (JVM) that every implementation must abide by. While it does not describe the implementation of the VM used in JavaOS for Business, it does provide a framework for discussing the details of the Java VM.
- The JDK 1.1.6 reference documentation describes the different packages used by applets to support user-interface and connectivity. This documentation is available on the JavaSoft web site (`http://www.javasoft.com/docs/index.html`).

How This Book Is Organized

The JavaOS for Business driver book is organized as a practical tutorial. The focus is on creating drivers from class templates and describing how drivers implement interfaces to perform I/O.

The book begins with basic scenarios that describe how JavaOS for Business building blocks (standard classes and interfaces) are arranged to create device drivers of different types. Throughout the tutorial, a sample network driver (PC3Com) is used to illustrate:

- how real-world drivers are structured
- how drivers are packaged for installation in the operating system
- how drivers function
- how bus managers provide low level memory and interrupt services to drivers

The use of sample (annotated) driver code helps to focus the discussion on actual--rather than academic--driver issues. The rationale behind this approach is that the specific object-oriented mechanisms that underlay one JavaOS for Business driver can be extended to create other JavaOS for Business drivers.

To a certain extent, the approach taken in this tutorial reflects a design principle inherent in the Java language: *expand on a pre-defined template base to implement custom methods in concrete ways.*

The PC3Com driver is our pre-defined template base. The skill set acquired in appreciating the mechanics of an actual, working driver is the skill set needed to create new drivers for custom use. Just as Java code is meant to be *reusable*, the methodology described in this tutorial is meant to be used and reused when creating JavaOS for Business drivers. In short, the premise that underlies this book is: *read once, write drivers anywhere.*

Chapter 1, Driver Scenarios, provides an introduction to JavaOS for Business drivers and describes ways in which drivers can be built from component parts.

Chapter 2, Driver Structure introduces a sample PC3Com driver and uses this to describe how classes and interfaces make up fundamental driver structures.

Chapter 3, Driver Packages describes how drivers are installed and loaded into the JavaOS for Business system.

Chapter 4, Driver Operations describes the actions performed by the PC3Com driver. This chapter provides annotated and complete PC3Com driver sample code.

Chapter 5, Bus Manager Services describes the memory and interrupt services provided to drivers by bus managers.

Table of Contents

List of Figures

1

Driver Scenarios

New computing appliances such as web terminals and NCs are managed with a stand-alone Java centric operating system known as JavaOS for Business. Loadable device drivers, written in Java, using memory and interrupt classes to abstract device hardware, are now an important, platform-independent part of the JavaOS for Business software.

This chapter presents a high-level overview of various driver types and examines the methodologies drivers employ to perform object-oriented I/O. In order to give a comprehensive view of driver types, SCSI drivers are included in this presentation, even though they have not yet been fully implemented in JavaOS for Business.

Introduction

In the PC, MAC, and UNIX worlds, writing device drivers has always been considered a complex and esoteric system programming task. Intimate knowledge of device registers, interrupt lines, memory management, thread context switching and so on are required, since drivers are expected to physically manipulate device hardware. Moreover, because drivers are actually part of the operating system kernel, extreme care must be taken to avoid system crashes. All this makes traditional C language driver writing both a challenge and a chore.

JavaOS for Business presents an entirely new model for writing drivers, different in kind as well as in degree from past driver paradigms. Taking advantage of inherent object-oriented capabilities, JavaOS for Business encapsulates much of the low level hardware detail in abstract classes and interfaces, providing developers with a streamlined, layered methodology for writing drivers in Java.

Role of Drivers in JavaOS for Business

The role of drivers in JavaOS for Business is to perform I/O. Pure and simple, that is all drivers do. Drivers implement class and interface methods to ensure the timely and efficient transport of data and control information throughout the JavaOS for Business system.

Driver Types

JavaOS for Business device drivers are dynamic system service modules--that is, discrete objects that plug into the operating system to perform specialized or generic I/O. Two types of drivers are present: logical and physical. Logical drivers may be stacked on top of one another or on top of a physical driver. Physical drivers connect directly to the underlying device hardware.

Driver Capabilities

Drivers implement bus manager interfaces to obtain memory, address, and interrupt information. Many drivers also extend device manager classes, which provide common functionality for drivers by device category. This includes SCSI drivers, audio and video drivers, network drivers and so on. In addition, drivers are sometimes characterized by the manner in which they transmit data to perform I/O. Serial and parallel drivers, block drivers, and other functional driver types are included in this group.

Abstracting out these capabilities, JavaOS for Business drivers facilitate I/O by performing one or both of the following services:

- Data formatting
- Device control

Consider the case where an application program wants to send data to a fax device. The application data is formatted in an array of bytes and the fax is connected to a serial port on the local machine. The I/O takes place as illustrated in the following figure:

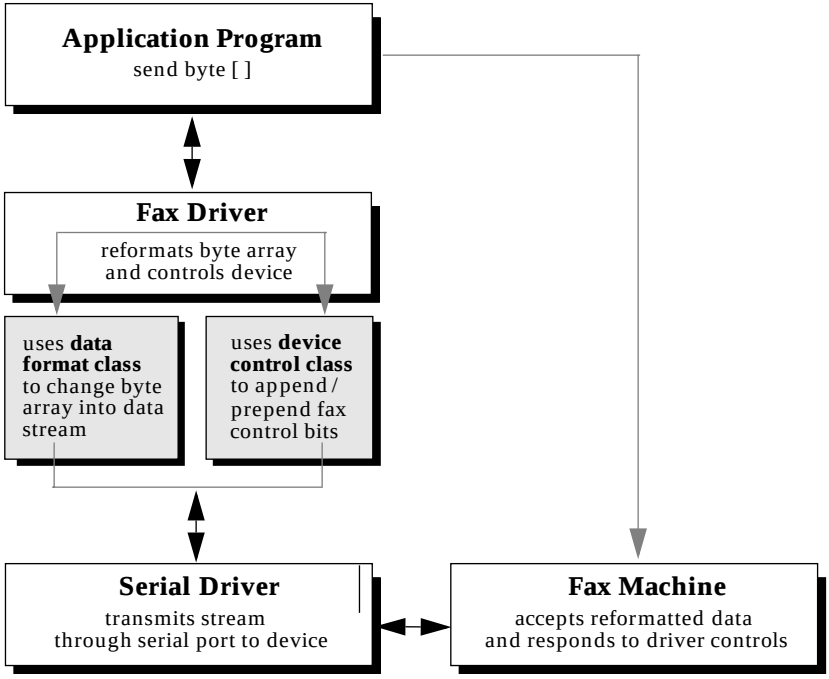

FIGURE 1-1 High-level view of driver I/O

A simplified view of the *data formatting* and *device control* performed by the Fax driver classes is illustrated below:

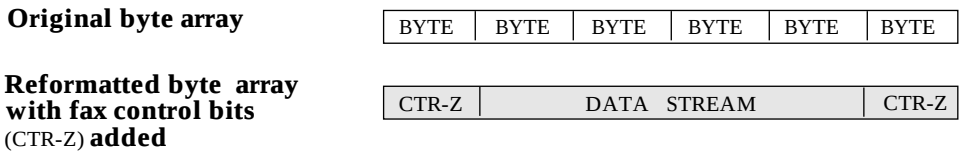

FIGURE 1-2 Reformatted byte array

The behind-the-scenes processing is actually more complex than this--various interfaces, bus managers, and device managers have been left out to streamline the flow; nonetheless, the diagram captures the base framework for all driver transactions. Depending on I/O concerns, application demands, and device

characteristics, the picture may expand dramatically or shrink marginally. But the essential driver I/O paradigm remains intact--JavaOS for Business drivers facilitate data transmission by providing data formatting, device control, or both.

Driver I/O Scenarios

To ground the abstract driver model just presented, this section provides a series of real-world examples of device behavior and interface types used to access a device or bus. Specific driver *families* and various types of *bus managers* and *bus types* are introduced. Scenarios are included for the following drivers:

- SCSI Drivers: All in the Family
- PCI Drivers: Single Parent Model
- SCSI on PCI: Adaptive Behavior
- Block Storage Drivers: Extended Family
- SCSI and Block Storage Families

In addition to these specific driver scenarios, the use of driver class templates is discussed and a high-level overview of the JavaOS for Business driver framework is illustrated.

SCSI Drivers: All in the Family

Drivers, like people, are sometimes identified by family group. A driver family consists of a device manager and one or more device drivers. The SCSI family, for example, is composed of the SCSI manager and one or more SCSI drivers. The SCSI family maintains relationships with other SCSI objects, such as bus adaptor devices and a SCSI bus interface. The SCSI family portrait looks like this:

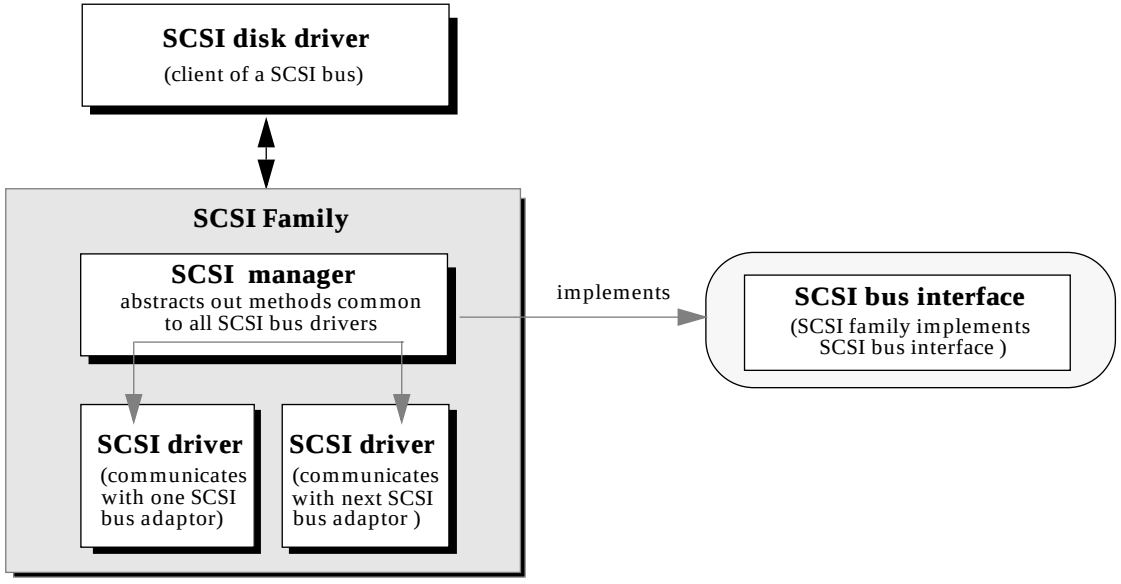

FIGURE 1-3 SCSI family

Here the SCSI family implements a SCSI bus interface to transact business with a SCSI disk driver, which is considered a client of the SCSI bus and is presumably acting on behalf of a file system to satisfy an I/O request.

Each driver family in JavaOS for Business publishes objects that implement a public I/O interface. Because JavaOS for Business makes full use of Java-language object encapsulation, the I/O resembles nothing more than narrowly-defined data flowing from one SCSI object to another. Since Java objects are modular and flexible, JavaOS for Business driver operations are also modular and flexible.

PCI Family: Single Parent Model

Within the SCSI family a clear division of labor was evident, with the SCSI manager abstracting out common driver functionality, and leaving adaptor drivers to deal with specific device implementations. The PCI family, however, operates on a different premise. Call it the single parent model, since PCI managers and drivers are essentially one and the same, exhibiting no real subdivision of labor. The PCI family picture looks like this:

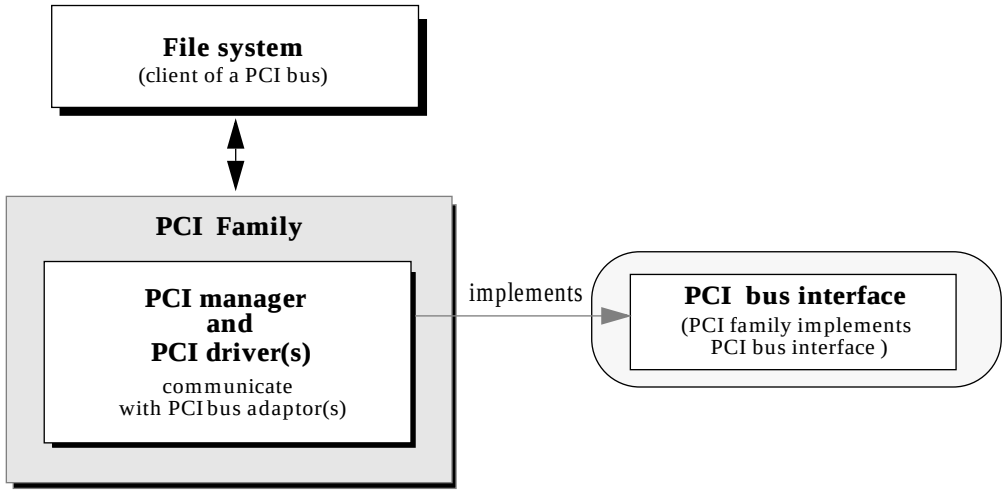

FIGURE 1-4 PCI family

Here a file system object is the client of a PCI bus and the PCI family publishes objects that implement the PCI bus interface. Business is conducted by the PCI *all-in-one* driver/manager. Because the PCI specification mandates the use of standard adaptor hardware, there is no need for multiple, implementation-specific PCI bus drivers. In the PCI family, drivers are embedded within managers. No external division of labor is apparent.

Family Values

The rationale behind this two-family approach to bus/driver organization derives from practical and historical differences in bus types: SCSI buses traditionally deal with I/O in an *ad hoc*, case-specific fashion, while PCI buses deal with I/O in a standardized, deterministic manner. JavaOS for Business drivers are flexible and modular enough to accommodate both models. Because JavaOS for Business eschews a monolithic approach to driver writing, JavaOS for Business drivers are able to conform to the framework of the bus architectures they traverse.

The following table tells the story in more detail.

TABLE 1-1 Tale of two bus types

Bus Type	Model	Physical Mode	Rationale
I/O	SCSI/ USB	external cable	The SCSI family is defined as a split-level family, composed of a SCSI manager and multiple SCSI drivers. Each driver is matched to a specific SCSI implementation involving a specific SCSI bus adaptor chipset. The SCSI family is characterized by a clear separation of powers. On the one hand, the SCSI manager abstracts the *chipset-independent* functions of the SCSI device--in other words, those functions common to all SCSI devices regardless of implementation detail. The SCSI adaptor driver, on the other hand, defines the *chipset-dependent* functions that control the hardware on the SCSI device. The USB family is also defined along the split-level lines of the SCSI family. It follows the SCSI single-manager/multiple-driver model. Like the SCSI family it is characterized by a chipset-independent top-half (the manager) and chipset-dependent bottom half (the driver).
Expansion	PCI/ ISA	slot in machine	The PCI bus family is a single-parent family, where the PCI driver is actually embedded in the manager. Because all PCI devices must implement a standard set of configuration registers defined by the PCI specification, there are no chipset-specific adaptor drivers required. The manager and driver are consequently folded into each other, leading to the *all-in-one* manager/driver model provided by JavaOS for Business. Because ISA buses operate within the exclusive framework of x86 platform architecture, they do not deviate significantly in implementation detail. No purpose is served by differentiating between managers and drivers.

SCSI on PCI: Adaptive Behavior

Multiple bus bridges can be constructed using adaptors connected to a PCI bus. Since PCI supports multiplexed bus connections, I/O and expansion buses may intersect at the junction of one or more bus bridges.

Strictly speaking, an expansion *bus bridge* is a chipset device that implements the protocol of an expansion bus. As described earlier, expansion bus bridges are managed by expansion bus managers, such as the PCI bus manager.

I/O bus adaptors often resides on expansion buses. For example, many SCSI adaptors are designed as PCI devices. In this circumstance, drivers assigned to a SCSI adaptor device are paired with a PCI expansion bus manager. The SCSI adaptor driver uses the PCI bus manager to gain device memory access and interrupt processing support.

JavaOS for Business encapsulates much of the complexity of this bus bridge behavior within respective I/O and expansion bus families. In practice, bus families may be stacked on top of each other to create a seamless transport mechanism, as illustrated below:

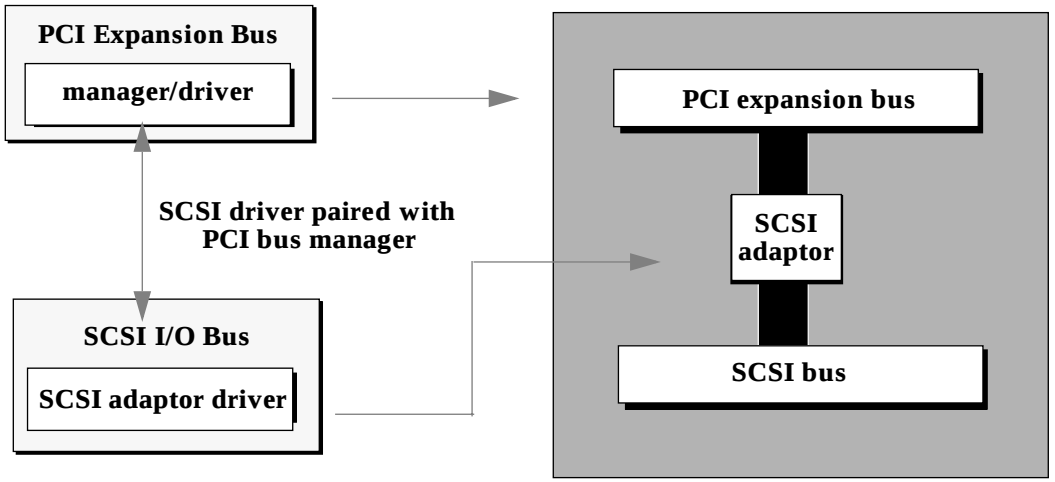

FIGURE 1-5 PCI to SCSI bus bridge

In summary, it is worthwhile noting that expansion bus managers insulate the operating system and device drivers from platform and chipset-dependent implementations of the bus bridge. Moreover, the set of all I/O buses, expansion buses and various bus bridges collectively make up the JavaOS for Business *bus transport layer*, an infrastructure upon which driver class implementation of *data formatting* and *device control* routines takes place.

Block / Partition Drivers: Extended Family

A block device is managed by a block storage driver, which is a member of a block storage family. The block driver is paired with a block storage manager because there is coordination required outside the realm of the driver. For example, block devices are partitioned into file systems. The block storage manager must examine all *disk* devices capable of being partitioned, including floppy disks and CD-ROMs. To do so, the manager goes outside the confines of its immediate family to implement a *partition interface.* The block storage manager uses the interface to look for file system partitions. For each partition found on the physical device, a virtual disk device is created.

Furthermore, as part of normal operations, the block storage family implements a *block storage interface.* This interface contains methods that read and write blocks within the partition.

The block storage family consequently supports *two kinds* of JavaOS for Business interfaces. The *partition interface* provides methods that deal with the type of file system contained in the partition, such as NT or UNIX. By implementing the partition interface, the block manager ensures that data destined for a file system partition is formatted appropriately. Moreover, the block storage family may be able to re-format partitions or to indicate if a partition is part of an even larger partitioning scheme like a RAID topology.

In effect, the block storage family takes on the added responsibility of formatting the data destined for the device so that the data is received without complaint. The block storage family then implements the *block storage interface* to convert file system requests into block partition addresses so that blocks may be written and / or read within the various partitions. This process is illustrated in the figure below:

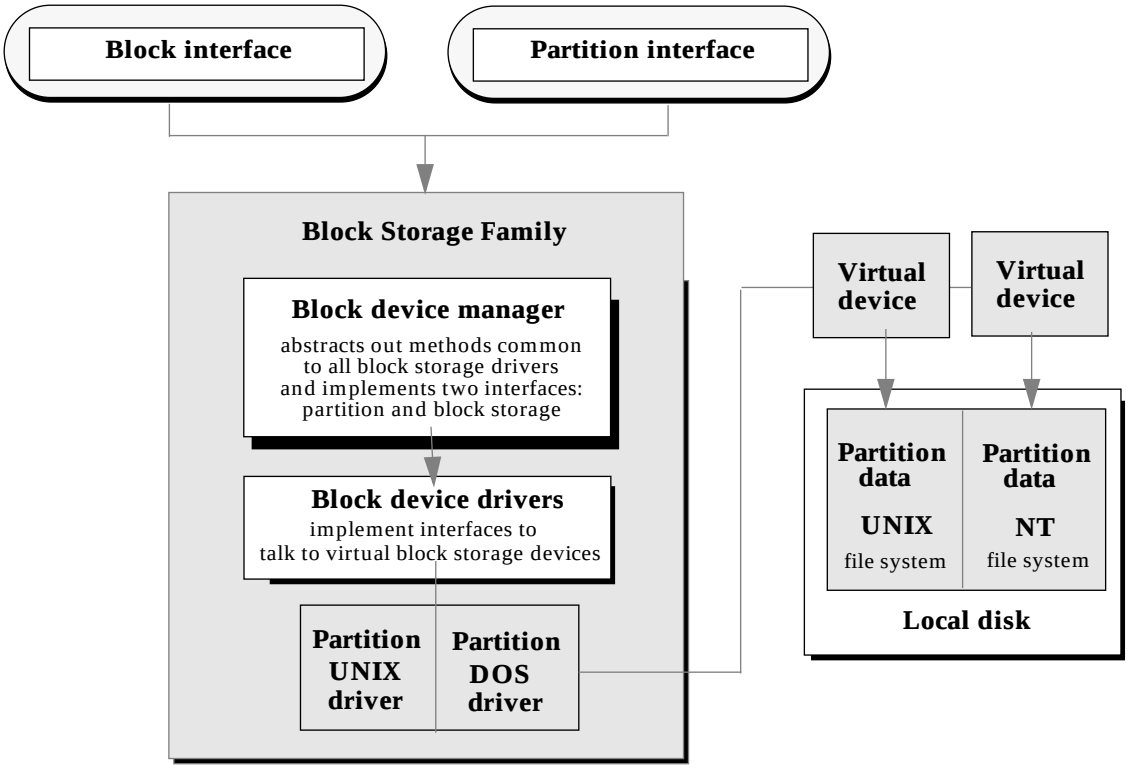

FIGURE 1-6 Block storage family

To accomplish this read and write activity, the device driver is defined in Java as:

```
public class DeviceDriver implements Partition, BlockStorage {...}
```

SCSI and Block Storage Families

Driver families often work together to achieve a common I/O goal. To present a more elaborate driver scenario, the following diagram illustrates a path where a file system client wants to read a block of data from a partition on a locally attached disk that is connected to a SCSI bus.

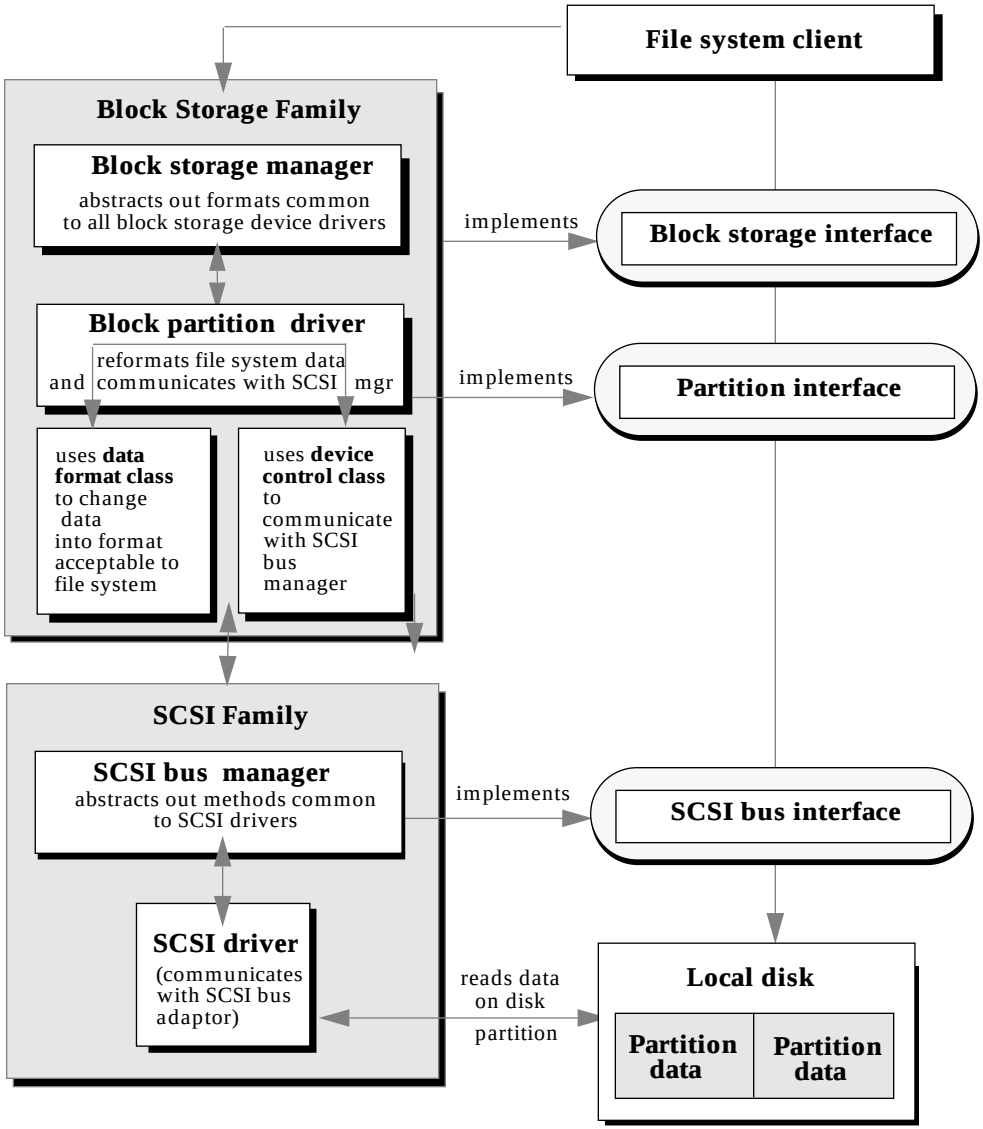

FIGURE 1-7 SCSI and block storage families

Two sets of device families are needed to service the I/O request. The first device family is the block storage family. The block storage family publishes block storage interface objects. The interface is implemented by the block storage manager and its set of drivers. The block storage family in turn becomes a client of the SCSI family. The SCSI manager publishes SCSI bus interface objects. Method calls to the SCSI interface object are handled by the manager and its set of SCSI bus instance drivers.

The device drivers used in this example are defined in Java as:

```
public class DeviceDriver implements ScsiBus
public class DeviceDriver implements BlockStorage
```

Driver Class Templates

The I/O system components discussed so far can be arranged in a variety of ways to address a nearly limitless combination of clients, devices, interfaces, and buses.

JavaOS for Business encourages additions to the framework to extend the reach and power of the operating system. To this end, base classes are provided with the system that act as *templates* for various kinds of drivers and managers.

The following figure shows how system-provided templates are leveraged into all component aspects of the JavaOS for Business driver framework.

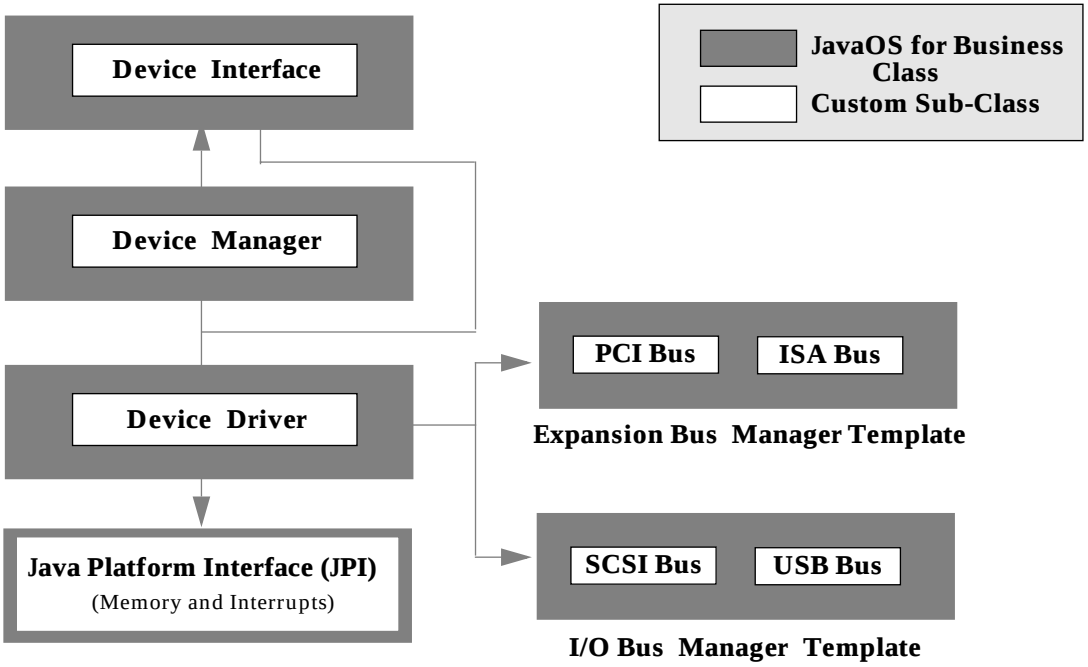

FIGURE 1-8 Driver templates

Java Driver and Platform Interfaces

This figure introduces the Java Platform Interface (JPI), a platform-dependent interface layer that abstracts the underlying hardware, allowing JavaOS for Business drivers to be platform-independent and written entirely in Java. The JPI works in concert with the Java Driver Interface (JDI) to provide a robust, secure, and loadable driver architecture. The complete driver architecture is presented in the next chapter.

Driver Framework Summary

To summarize (and simplify) the main driver framework presented so far, a driver data-type diagram presents several possible driver / manager / bus layering combinations for JavaOS for Business drivers.

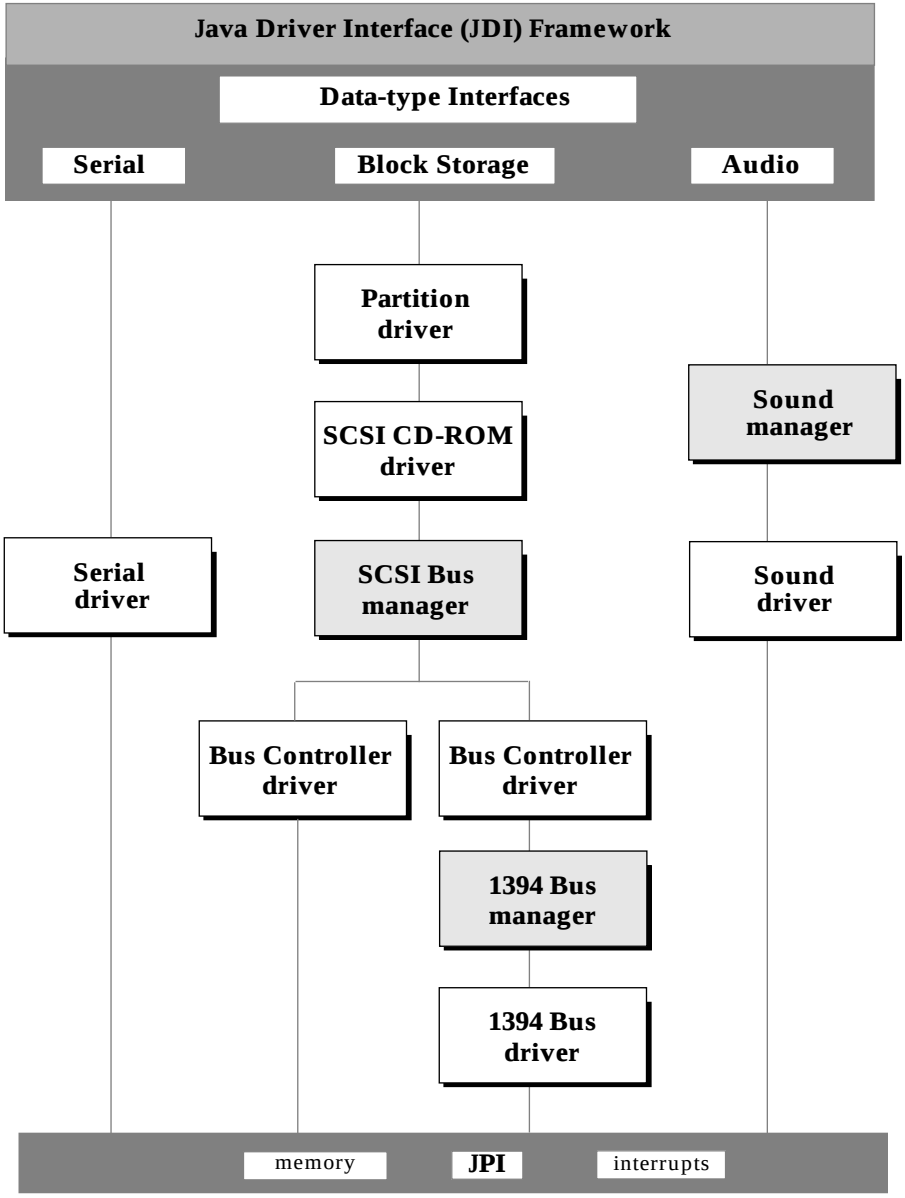

FIGURE 1-9 Driver, bus, and manager layering

Definition of Terms

This section provides a list of JDI terms for defining some basic concepts of the I/O class framework discussed so far.

Client

A *client* is any Java class that is using the public services of the JDI. The class may exist within or outside of the JDI class framework. Examples include applications like HotJava and browser applets as well as JDI device drivers layered upon other device drivers.

Device

A *device* is a abstract representation of a physical or virtual hardware entity. Physical hardware devices are tangible and typically attached to an I/O bus. The JDI relies on the Java System Database and the JBI to obtain the set of physical devices discovered on the platform or assigned to a platform by an administrator. The set of physical devices is published via the JSD as a *tree* of devices. See the JSD and JBI specifications for more device tree details.

Virtual devices on the other hand, are purely software fabrications. Examples include a RAM disk, or a disk partition. Virtual devices are appended to the physical device tree dynamically as discovered.

Device Driver

A *device driver* is a Java software component designed to manage one or more instances of a device. A device driver is composed of a group of classes arranged in packages. One or more of these packages must implement JDI-compliant interfaces, including the standard *Device* interface. A device driver can also be a client of another device driver using JDI services.

A device driver fulfills the terms of the contract defined by one or more JavaOS for Business interfaces. All device drivers are defined as a descendent of the standard *device driver class*, and are further characterized by the set of specific interfaces implemented.

The parent device driver class is defined as follows:

```
class DeviceDriver implements Device {...}
```

Descendant driver classes are defined:

```
class ScsiDeviceDriver implements ScsiDevice {...}
class SerialDeviceDriver implements SerialDevice {...}
```

The ScsiDevice interface is defined as:

```
interface ScsiDevice extends Device {...}
```

The SerialDevice interface is defined as:

```
interface SerialDevice extends Device {...}
```

Driver Types

In general, drivers can be classified as either:

- logical drivers
- physical drivers.

Logical Drivers

Logical drivers are virtual drivers. They talk to other drivers, rather than to hardware, and they do not require a bus manager. Logical drivers open up a layer in the device tree one or more levels above a physical driver. There may be many logical drivers on top of one physical driver. Logical drivers are platform independent.

Physical Drivers

Physical drivers talk to devices using the services of a bus manager, which calls into the hardware on behalf of the physical driver. Bus managers encapsulate hardware dependencies, allowing physical drivers to remain at one remove from platform devices. Physical drivers are consequently platform *independent*. For the same reason, bus managers are platform *dependent*.

Device Manager

A *device manager* is a Java software package providing clients a common interface to a *category* of devices. For example, all Small Computer Systems Interface (SCSI) devices belong to the SCSI device category. The SCSI device manager (or *SCSI manager* for short) provides a unified SCSI interface for all flavors of SCSI bus controller.

A device manager also provides device specific services to a set of device drivers. For example, the SCSI manager provides SCSI specific services to the set of active SCSI bus device drivers.

Not all device drivers however are paired with a manager. Some devices are easily managed with stand-alone drivers. For those drivers paired to a manager, *the manager contains all the functionality common to the category of device.*

The parent device manager class is defined as follows:

```
class DeviceManager implements Device {...}
```

The scsi device manager class is defined as follows:

```
class ScsiDeviceManager implements ScsiDevice {...}
```

JavaOS for Business supplies a default device manager class that is the initial owner of all devices in the system. The default device manager implements the standard device interface for all devices in the system. Through-out the remainder of the document, the term “Device Manager” refers to the default device manager supplied by JavaOS for Business.

As each device is characterized, its ownership and implementation responsibility is transferred to either a stand-alone device driver or another more specialized device manager. Either way, a driver or a manager implements some form of the standard Device interface. Specialized device managers can in turn transfer device ownership to a driver associated with that specific manager.

If a driver gives up ownership of a device, the default or specialized manager regains control. If the device is removed from the device tree, all code assigned to manage that device may be unloaded from the system as well.

Device Family

An *device family* is the collective term for a *device manager and its set of device drivers.* For example, the SCSI family is the term used to name the SCSI manager plus the set of active SCSI drivers. A device family is also associated with one or more public I/O interfaces.

The *generic device family* is the collection of stand-alone device drivers and the default device manager.

Bus

A device that allows other devices to connect themselves to the platform.

Bus Manager

Bus Managers insulate device drivers from platform and bus dependencies by providing low-level system services such as memory allocation and interrupt processing. This allows drivers to be written entirely in Java and remain architecture neutral. It also simplifies the task of driver writing by removing bus management concerns. For driver writers, the result is streamlined access to system resources, reduced code overhead, and faster development cycles.

2

Driver Structure

A range of possible driver types was presented in Chapter 1--not all of which are fully implemented in JavaOS for Business--to give an overview of the way drivers are constructed and the operations they perform.

The remainder of this documentation is tutorial in nature and focuses on a complete, functioning JavaOS for Business driver, the PC3Com Ethernet driver of the Network Driver family. Sample PC3Com driver code is used to illustrate both the structure of the driver and specific internal driver operations. A hands-on approach is taken to define the fundamental classes, interfaces and methods of the driver code.

This tutorial is written with the assumption that readers will consult the accompanying Reference Guide documentation for a deeper understanding of the concepts presented here.

In this chapter, the structure of the PC3Com Ethernet driver is presented. Sample code from the PC3Com driver is used to demonstrate class extensions and interface implementations. PC3Com is an Ethernet driver responsible for sending packet information to a network server.

PC3Com Overview

In JavaOS for Business, device drivers are simply system services that perform I/O. Like any other system service, drivers can be dynamically loaded and unloaded on demand.

For convenience, sample code for a driver like PC3Com can be grouped into two categories:

- code that connects the driver to the operating system

- code that connects the driver to the underlying device

OS Connection Code

OS connection code covers such activities as creating a driver instance, starting a driver service, requesting system resources, advertising driver services to the system, and performing device-specific operations in concert with other components of the system. All of these driver operations are performed using a flexible, but predefined, set of APIs.

Device Connection Code

Device connection code consists of driver-specific, device-specific operations that perform internal housekeeping and keep track of the *state* as well as the *status* of the device and driver. These routines are device-specific. JavaOS for Business provides *some* pre-defined device-specific interfaces that driver writers can implement. However, in cases where no predefined APIs are provided, driver writers may need to create custom device interfaces.

A driver writer may choose, for example, to use methods like *read* and *write* to retrieve and store device data, but there is no standard nomenclature required when naming device-specific methods. Naming conventions are a matter of personal choice. Driver writers are, so to speak, left to their own devices to work out the details.

As a consequence, the PC3Com driver code in this document is *only* analyzed in terms of OS connection code, since that is the code used to interface with the rest of the JavaOS for Business framework.

PC3Com is the driver for the 3Com 3C905 Fast EtherLink XL PCI adapter. It also supports the 3Com 3c590 adapter.

JavaOS for Business Framework

The PC3Com driver interfaces with several important components of the JavaOS for Business system I/O framework, which consists of system components grouped under the JavaOS Device Interface (JDI) such as device drivers and managers, bus managers, and the JavaOS Platform Interface (JPI), which bus managers use to provide memory and interrupt resources to drivers. To request system resources, for example, drivers typically use methods defined in the JDI. Bus managers, acting on behalf of drivers, may subsequently use methods defined in the JPI.

For a physical driver like PC3Com connected to a device on a PCI bus, an expansion *bus manager* acts as the resource provider. The bus manager is a piece of trusted driver code with the ability to access low-level system resources beyond the reach of physical and logical drivers. The bus manager invokes the JPI to obtain memory and interrupt resources. By intermediating device driver access to low-level system resources, bus managers ensure driver portability--drivers are written entirely in Java--and protect sensitive system resources from unwanted access.

JavaOS for Business drivers that handle interrupts need to create an interrupt source and register the source to the system via the bus manager. When a driver starts up it negotiates with the bus manager for its interrupt object and this object is plugged into the interrupt source tree. The tree is a hierarchy of interrupt source objects, where the root of the tree represents the platform CPU.

The driver interrupt source class inherits from the `DeviceInterruptSource` class.

Memory and Interrupt Methods

The PC3Com driver calls methods in `ExpansionBusManager` to request system resources from the bus manager. These resources include:

- memory descriptors
- memory constraints
- allocated memory
- allocated interrupts
- allocated subranges of memory

Bus managers use *memory descriptors* when calculating device addresses on a bus (or on a series of bus bridges). Memory descriptors do not represent actual allocated memory; instead they describe device *requirements* for memory and can be thought of as secure memory tokens. They are passed around by the bus manager and eventually converted into real memory.

Alignment requirements for memory constraints may involve byte swapping or data ordering issues, that is, processor-specific constraints on endian-ness or strict rules on the arrangement of data.

The requirements for memory and interrupt classes are described in greater detail in Chapter 5, Bus Manager Services.

Services

The JavaOS Service Loader (JSL) manages the loading and unloading of both applications and system software components known as *services*. Services are most easily described as typed libraries of Java code. A service can encapsulate various

types of operating system functionality; JavaOS for Business services, for example, include device drivers, network protocol stacks, and file systems. Clients, drivers, and managers are all implemented as services.

Services are typically bundled into packages. The JavaOS for Business networking subsystem makes extensive use of JDI service packages to implement network drivers and protocols. The `PC3Com` driver implements a set of interfaces associated with both the JavaOS System Database (JSD) and the JavaOS Service Loader (JSL). The JSL invokes `Service` class and `ServiceInstance` interface methods to perform PC3Com driver instantiation, initialization, loading, unloading, and so on.

All drivers must provide support to the JSL for static initialization and subsequent instance creation.

Service Class Methods

The following methods are defined by the `Service` class and called by the JSL to manage or interact with the `PC3Com` driver instance.

```
public static void start(Entry businessCard) throws ServiceException {
        System.err.println(PC3Comstart, businessCard=+businessCard);
        businessCard=businessCard;
    }
public static void stop() {}

public static void suspend() throws ServiceException {}

public static void resume(Entry e) throws ServiceException {}
```

The JSL calls `start()` immediately after loading a service. Drivers, such as `PC3Com`, can use this method to do one-time device initialization. The parameter passed in is the PC3Com business card entry located in the software namespace of the JSD. The business card contains the driver configuration properties. Business card, JSL, and JSD operations are described in greater detail in Chapter 3, Driver Packaging.

Sample code for the `ServiceInstance` method invoked by the JSL to instantiate the PC3Com driver is examined later in the *Interface* section of this chapter.

Class Hierarchy

The PC3Com network driver is part of a class hierarchy that includes a generic driver, a network driver, and the PC3Com driver. The class files comprising this hierarchy are:

- `DeviceDriver.java`
- `NetworkDeviceDriver.java`

- PC3Com.java

The DeviceDriver class is extended by the NetworkDeviceDriver class, which in turn is extended by the PC3Com class.

Device Driver Class

```
public abstract class DeviceDriver extends Service implements Device
```

This is the class all JavaOS for Business drivers must extend. It is the base class extended by NetworkDeviceDriver. It provides generic *open/close* methods as well as a set of open device handles and routines for managing handle operations. It also provides methods for obtaining the device name, the set of actions that can be performed by the device, and so on. It is a *template* for use by driver developers and implements the Device interface.

Network Device Driver Class

```
public abstract class NetworkDeviceDriver extends DeviceDriver
    implements NetworkDevice
```

This class passes data and control information to and from the transport layer of the network protocol stack. It extends DeviceDriver and is extended by the PC3Com class. As a generic network class, it obtains device-dependent network packets and provides formatting methods that make the packets suitable for output. In addition it provides methods for adding/managing Multicast addresses, for obtaining interface status, and (if appropriate) returning boot configuration information. It is a logical driver and implements the NetworkDevice interface.

PC3Com Class

```
public final class PC3Com extends NetworkDeviceDriver
    implements EthernetDevice,EthernetDriver,ServiceInstance
```

This is the lowest level class in the hierarchy, the physical driver responsible for managing the device. It includes ServiceInstance methods called by the JSL to obtain driver services and implements the EthernetDriver interface to add/delete Multicast address or enable/disable promiscuous mode.

Since the PC3Com class inherits from both the NetworkDeviceDriver and DeviceDriver classes, the PC3Com driver actually represents the *union* of all three classes taken together.

For a complete listing of the functions associated with the PC3Com driver, refer to Chapter 4, Driver Operations, where complete driver code is reviewed.

Interface Hierarchy

The PC3Com driver is the bottom rung of an extended device interface hierarchy that includes the following interfaces:

- `Device`
- `NetworkDevice`
- `EthernetDevice`
- `ServiceInstance`

Device Interface

```
public interface Device
```

The base `Device` interface defines a set of generic methods that all devices support. Every device in JavaOS for Business must implement this interface. Subsequent interfaces are created by extending the `Device` interface.

The `Device` interface allows clients to get the name of this device or to retrieve other information such as which driver, if any, is currently managing the device. Additional information concerns the actions possible with the device and the state of handles operating on the device.

The following methods are provided:

- `getName`
- `getDeviceActions`
- `getOpenDeviceHandles`
- `getCurrentOwner`

NetworkDevice Interface

```
public interface NetworkDevice extends Device
```

`NetworkDevice` defines an interface that drivers implement to obtain device dependent IP packets for network output. It also provides methods to obtain a boot configuration object, to summarize in report format interface status, and to determine the maximum length of transmission packets (Mtu) and packet header information. The following methods are provided:

- `getTransmitPacket`
- `output`
- `getConfig`
- `report`

- getMtu
- headerHint

EthernetDevice Interface

```
public interface EthernetDevice extends NetworkDevice
```

EthernetDevice defines an interface that drivers implement in order to peek and poke at low level device register information and to establish a filtering mechanism for network broadcasts.

- addMulticast
- deleteMulticast
- enablePromiscous
- disablePromiscous

ServiceInstance Interface

```
public interface ServiceInstance
```

ServiceInstance defines an interface with the following methods:

- createInstance
- deleteInstance

The ServiceInstance interface is used by the JSL to instantiate the PC3Com driver using the createInstance method. A new driver instance is returned. createInstance parameters describe:

- the device business card, which contains configuration properties for the device
- the logical name of the device
- the bus parent of the device
- the device alias in the JSD
- a cookie generated by the JSL that the driver can use as an instance ID

The createInstance method is called by the JSL to request an instance of a service, in this case the PC3Com driver instance. To provide instance creation and deletion capabilities, the driver must include a class that implements the ServiceInstance interface. The bundleInstanceClassName parameter of the business card contains the name of the class that provides this implementation.

Sample PC3Com code follows for `createInstance` and `deleteInstance` methods:

```
public static ServiceInstance createInstance(
        Entry businessCard, String logicalName, Object parent,
        Entry aliasEntry, Object cookie) throws ServiceException {

    System.err.println(PC3Com::createInstance, businessCard=
        +businessCard);
    System.err.println(PC3Com::createInstance, logicalName=
        +logicalName);
    System.err.println(PC3Com::createInstance, parent=
        +parent);
    System.err.println(PC3Com::createInstance, aliasEntry=
        +aliasEntry);
    System.err.println(PC3Com::createInstance, cookie=+cookie);

    // This implementation actually creates a new instance of the driver.
    if (pc3Com == null) {
    try {
        pc3Com=new PC3Com((Bus)parent, businessCard);
        System.err.println(PC3Com::createInstance,
            + binding PC3Com instance into env, tk=+pc3Com);
        Env.bind(/dev/ether, pc3Com);
    } catch (Exception tex) {
        System.err.println(PC3Com::createInstance,
            + PC3Com() threw exception=+tex);
        throw new ServiceException(businessCard,tex.toString());
    }
} else {
    System.err.println(PC3Com::createInstance,
        + returning reference to existing pc3com=+pc3Com);
}
return pc3Com;
}
```

The remaining interface method of `ServiceInstance` is `deleteInstance`. It has a single parameter, a cookie, which represents the driver `instanceID`

```
public void deleteInstance(Object cookie) throws ServiceException {
   System.err.println(PC3Com::deleteInstance, PC3Com=+this);
}
```

The PC3Com driver implements one additional interface, `ExpansionBus`, which is covered in chapter 5, Bus Manager Services.

Class / Interface Relationships

The classes and interfaces detailed above are the component building blocks of the PC3Com driver. The logical arrangement of these components is illustrated in the following diagram, which establishes a class / interface matrix. Notice that:

- The *class* hierarchy is conveyed *vertically*, as each driver (except the topmost) extends the class directly above it.
- The *interface* hierarchy is conveyed *horizontally*, as each class implements one or more device interfaces.

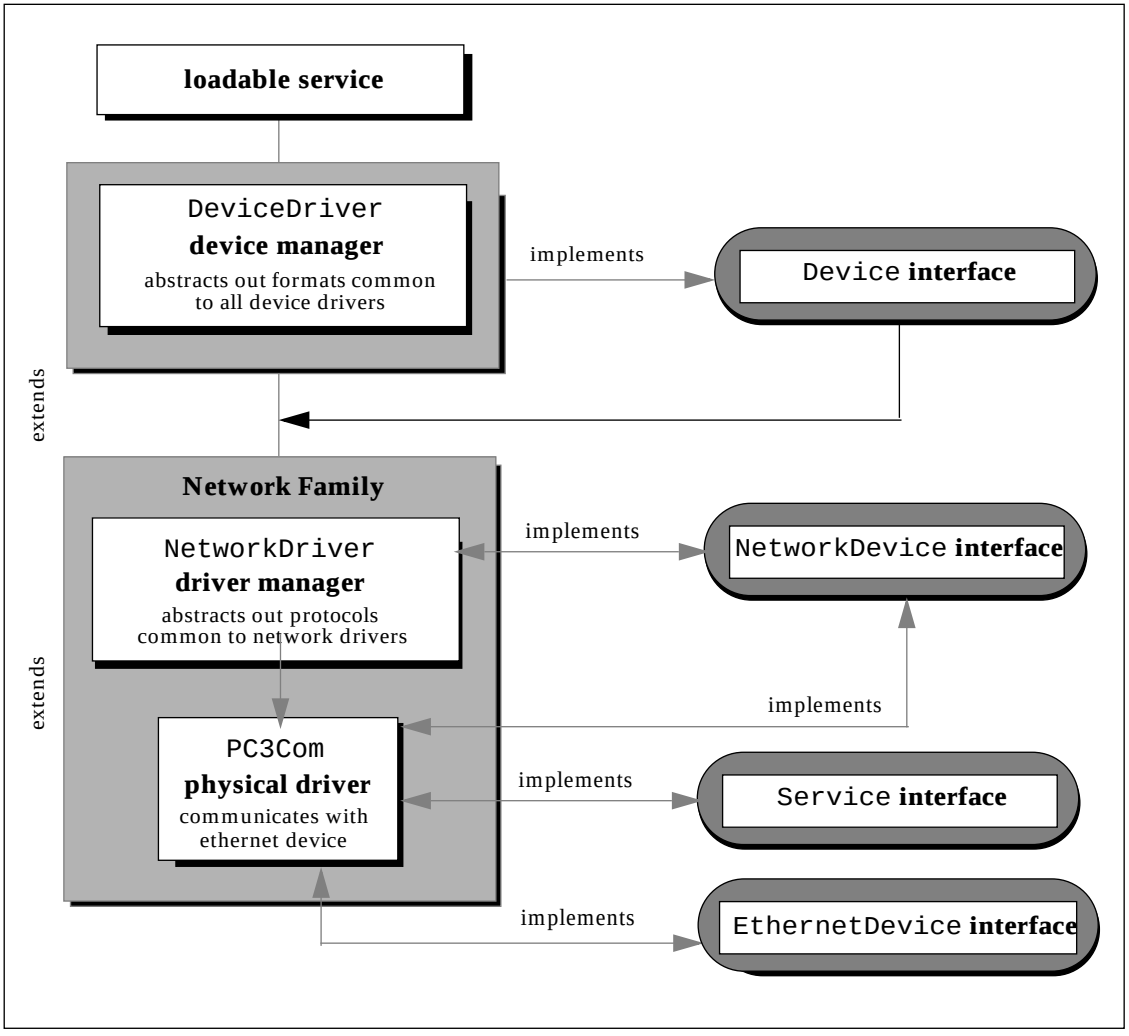

FIGURE 2-1 PC3Com class / interface diagram

PC3Com Source Listing

The following source tree listings display the classes and interfaces implemented as part of PC3Com driver operations:

```
#
# PC3Com code
#
services/src/PC3Com/sun/javaos/PC3Com/PC3Com.java
services/src/PC3Com/sun/javaos/PC3Com/PC3ComDmaPacket.java
services/src/PC3Com/sun/javaos/PC3Com/PC3ComPacket.java
services/src/PC3Com/sun/javaos/PC3Com/PC3ComInterruptSource.java

#
# Classes extended and implemented
#
src/javaos/javax/javax/system/jdi/Device.java
src/javaos/javax/javax/system/jdi/NetworkDevice.java
src/javaos/javax/javax/system/jdi/DeviceDriver.java
src/javaos/javax/javax/system/jdi/NetworkDeviceDriver.java
src/javaos/javax/javax/system/jdi/EthernetDevice.java
src/javaos/javax/javax/system/jdi/EthernetDeviceHandle.java
src/javaos/sun/sun/javaos/net/EthernetDriver.java

#
# Classes used to connect with JSL
#
src/javaos/javax/javax/system/services/Service.java
src/javaos/javax/javax/system/services/ServiceInstance.java

#
# Memory classes
#
src/javaos/javax/javax/system/memory/AccessibleMemory.java
src/javaos/javax/javax/system/memory/Address.java
src/javaos/javax/javax/system/memory/Address32.java
src/javaos/javax/javax/system/memory/AllocationException.java
src/javaos/javax/javax/system/memory/InvalidAddressException.java
src/javaos/javax/javax/system/memory/Memory.java
src/javaos/javax/javax/system/memory/MemoryConstraints.java
src/javaos/javax/javax/system/memory/MemoryDescriptor.java
src/javaos/javax/javax/system/memory/MainMemory.java

#
# Interrupt class
#
src/javaos/javax/javax/system/interrupts/DeviceInterruptSource.java

#
# Bus manager
```

```
#
src/javaos/javax/javax/system/jdi/ExpansionBusManager.java
src/javaos/sun/sun/javaos/pci/PCPCIBusManager.java
```

What is Ahead

In this chapter a blueprint was given for constructing drivers based on class and interface hierarchies. The `PC3Com` driver was used as an example. In the next chapter, procedures are given to package and configure the `PC3Com` driver so that it can be dynamically loaded into the client-side JSD.

3

Driver Packaging

The previous chapter presented a high-level view of driver structure. The next chapter presents a low-level view of driver operations. Between these two extremes lies the mundane, but crucial, task of loading and configuring a driver for the JavaOS for Business system.

Packaging a driver means charting the interaction among several JavaOS for Business components, including the JavaOS System Database (JSD), the JavaOS Configuration Tool (JCT), and the JavaOS Service Loader (JSL). These components are vital processing points that enable drivers to be wrapped so they can be recognized and loaded dynamically.

The purpose of this chapter is to illustrate the mechanics and functional flow involved in driver packaging. The JSD, JSL, and JCT are described only to the extent they impact driver packaging. For a more detailed view of these components, refer to the Reference Guide documentation.

Packaging the Driver

The JavaOS System Database is the central repository of driver registration and configuration information. Essential driver data is retained within the JSD in the form of *business cards*, which are created by driver writers and whose parameters provide: (1) self-configuring information to drivers and; (2) information about driver capabilities to other services, applications, and JavaOS for Business system components. A business card must be unbundled by the JCT and placed within the JSD before a driver can be loaded into the system by the JSL.

The processing required to prepare a device driver for system loading is referred to as *packaging the driver*. The major steps involved in this process are summarized and diagrammed later in this chapter. Some background information on the JavaOS for Business configuration process is necessary, however, before packaging can be fully understood.

NOTE: JavaOS for Business supports *both* JAR and ZIP files for driver packaging. However, the examples used in this chapter refer exclusively to JAR files as the package container.

Service Packaging Configuration Issues

There are two types of loadable services in JavaOS for Business: static and dynamic. Static services are loaded by the system booter when JavaOS for Business first starts-up. The booter is responsible for loading services such as drivers into the operating system during the boot process. This is a static operation, as the set of services loaded at boot time is defined up-front using the JCT. Alternatively, other services are loaded dynamically during runtime by the JSL. This occurs, for example, when non-boot devices are first opened.

When drivers are loaded by the booter, each driver is inserted into the ROM file system embedded within the OS. This embedded file system is used to bootstrap the OS. Drivers loaded at boot time must configure the JAR file such that the code for the driver is outside the confines of the JAR, as the booter will not read code from within the JAR.

When drivers are loaded dynamically by the JSL, the driver code itself can be included in the JAR file, along with the business card and other driver beans. The advantage of this is that the JAR file is a fully self-contained driver package, including a copy of the complete driver code.

The following procedure describes the steps involved in dynamically loading a driver package.

Creating and Installing a Driver Package

1. The developer writes a *business card* for the PC3Com driver, which contains configuration information essential to the JSD, JSL, JCT, and the driver itself.

2. The JCT is a server-side, GUI-based, system administration configuration tool that accepts JAR or ZIP files, parses the data, and places it into the *configuration* namesapace on the server JSD. In order for the JCT to recognize and install the `PC3Com` business card, it must be wrapped in a *configuration bean* and placed in a JAR file.

3. To create a JAR file containing configuration beans, several files are packaged together into `PC3Comcnfg.jar`

4. The `JAR` file is copied to the directory named `jars` at the top of JCT directory on the server. For example:

   ```
   cp PC3comcfg <top_jct_directory>/jars
   ```

5. The JCT parses the file and inserts business card data into the `software` namespace of the JSD, causing an `insertEvent` to be propagated.

6. On the client side, the JSL consumes the event and communicates with the server-side JSD so that the business card entry is sent down the wire and installed in the *software* namespace on the client.

7. The `PC3Com` business card contains a reference to the `PC3Com` driver code in the JAR. The JSL locates the driver class file and invokes the `createInstance` method to instantiate the `PC3Com` driver and load it on the client. The driver instance `pc3com` is returned by `createInstance`.

8. Once creation of the driver package is complete, the JSL uses the *tmp* namespace in the client JSD to keep track of business card activity.

This process is illustrated in Figure 3-1.

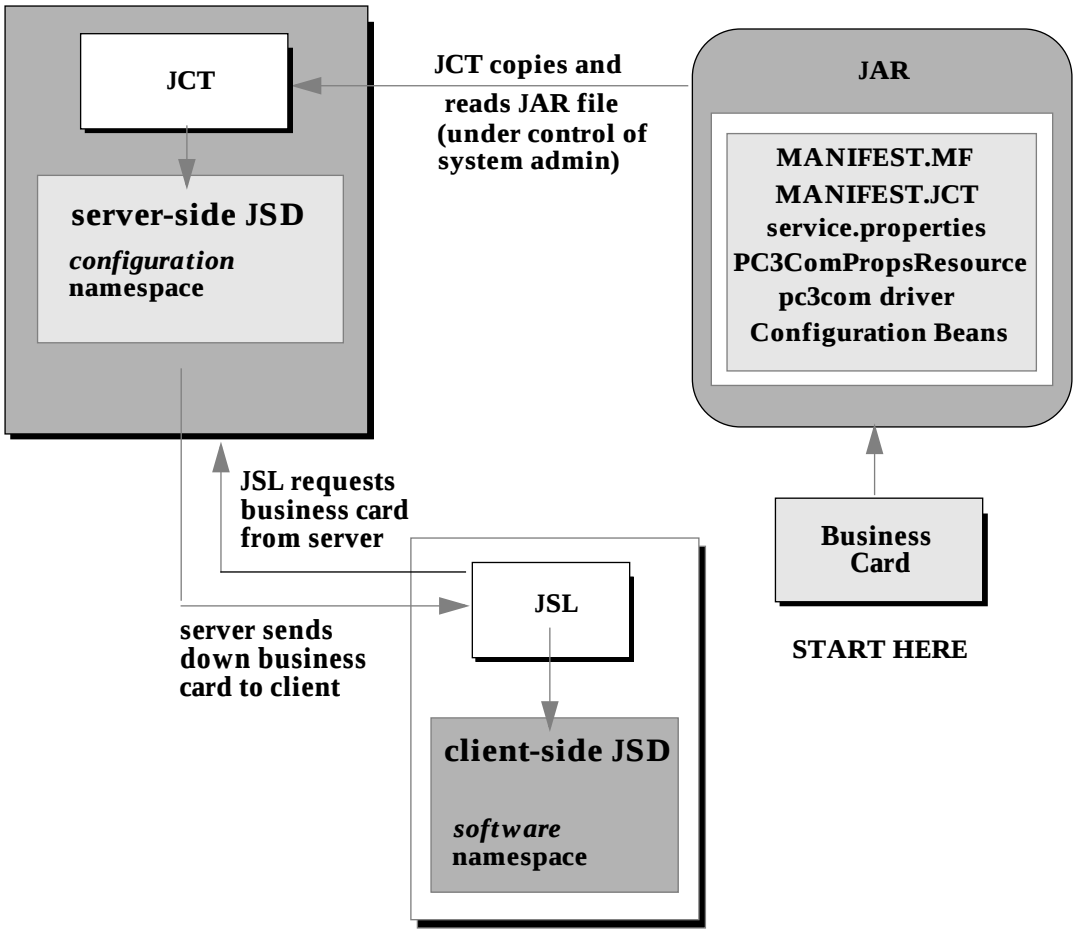

FIGURE 3-1 Driver packaging

Services and Business Cards

The `BusinessCard` interface is a set of configuration parameters comprised of both private and framework parameters. The private parameters are used by the driver to configure itself; the framework parameters are used by the JSL to discover, load, advertise, and instantiate the driver.

Each service is described by a class (within the service JAR container) called a *business card*. Business cards are useful sources of driver information because they:

- Name the driver
- Provide vendor and version information
- Advertise interfaces implementation by the driver
- Reference the bundle containing the driver code to load
- Provide configuration parameters to the driver

To advertise its services, a device driver (or any other system service) must have a business card. The business card is created by the driver developer to provide the configuration information necessary for the driver to retrieve the `Service` interface and advertise the interfaces it supports. Business cards are added to the server-side JSD by the JavaOS Configuration Tool (JCT), running on the same server.

JCT Configuration Beans

The JCT uses beans packaged in specially formatted JAR files to define the content, restrictions, and syntax of the JSD entries. Configuration beans provide a set of interfaces, classes, and property text files. There are specific rules for implementing configuration beans. These rules allow the JCT to accomplish three main tasks:

- Recognize and display the driver (or service) properties
- Insert the driver into the `configuration` namespace on the server-side JSD
- Allow the JSL to load the driver into the `software` namespace on the client-side JSD

`PC3Comcfg` Files

`PC3Comcfg` is the name of a JAR file created to package the business card as a configuration bean. `PC3Com` contains the following files:

- `PC3Comcfg/MANIFEST.MF`
- `PC3Comcfg/MANIFEST.JCT`
- `PC3Comcfg/mri/PC3ComPropsResource.properties`
- `PC3Comcfg/service.properties`

MANIFEST.MK

The MANIFEST.MK file provides direct identification of the configuration beans.

```
Name: com/ibm/joscfg/services/JSLBootProps.class
Java-Bean: True

Name: com/ibm/joscfg/services/JSLSimpleInstance.class
Java-Bean: True

Name: com/ibm/joscfg/services/JSLLoadProps.class
Java-Bean: True

Name: com/ibm/joscfg/services/JSLDevProps.class
Java-Bean: True
```

MANIFEST.JCT

The MANIFIEST.JCT file provides package-level attribute information to the JCT, enabling it to determine the JSD entry location for a bean. This information includes:

```
ResourceBundle=com/ibm/joscfg/PC3Comcfg/mri/PC3ComPropsResource
PackageType=SERVICE
JSDEntryName=/software/service/sun.javaos/PC3Com
JSDStoreFormat=ENUMERATE
EntryDisplayOrder=com/ibm/joscfg/services/JSLBootProps.class com/ibm/
    joscfg/services/JSLDevProps.class com/ibm/joscfg/services/
    JSLSimpleInstance.class com/ibm/joscfg/services/JSLLoadProps.class
META-Property=.BOOTTIME true

Name=com/ibm/joscfg/services/JSLBootProps.class

JSDPropertyMap=MCFBundleURL .BOOTSERVICE

Name=com/ibm/joscfg/services/JSLSimpleInstance.class
Name=com/ibm/joscfg/services/JSLDevProps.class
Name=com/ibm/joscfg/services/JSLLoadProps.class
```

Short descriptions of each MANIFIEST.JCT attribute follow.

ResourceBundle

Package-level attribute used as the bundle name to localize PackageName, PackageAbout and DisplayName

PackageName

Package-level attribute used to describe the overall contents of the JAR file. It is used by the JCT for display and selection.

PackageType

Package-level attribute used to describe the classification of the software.

PackageAbout

Package-level attribute used by the author to describe version, licensing and author information for the JAR file.

PackageHelp

Package-level attribute used by the author to provide help when configuring any bean.

JSDEntryName

Package-level attribute used to describe where the configuration information should be stored. This path represents the storage location for the bean in the JSD.

JSDStoreFormat

Package-level attribute used to describe the actual storage format for the configuration information being placed in location `JSDEntryName`.

EntryDisplayOrder

Package-level attribute used to describe the order in which the beans should be displayed in the `BeanUI`.

META-Property

Package-level attribute used to describe additional properties for this JSD Entry.

Name

Entry attribute used to describe the bean for which the attributes apply.

JSDPropertyMap

Entry attribute used to map bean property names to JSD entry property names.

PC3ComPropsResource.properties

PC3ComPropsResource.properties contains attribute information important to the JCT, such as:

```
PackageName=3COM Fast EtherLink XL PCI 3C905 Driver
PackageAbout= 3COM Fast EtherLink XL PCI 3C905 Driver Version 0.1
PackageHelp= \
```

Using the JCT, a GUI tool that displays the contents of the business card on screen so that properties can be manually edited by the system administrator.

- In the Service URL field, for example, specify the URL of the server where the 3COM Fast EtherLink XL PCI 3C905 driver for this platform or machine is installed. The file name PC3Com.jar cannot be changed, but protocol, server, and directory names may be changed.
- In the Matching name field, type the IEEE 1275 formatted device identifier
- In the Logical name field, type the name by which this driver will be recognized. This name is defined by the JSL.
- Press the Customize button to specify when the driver will be loaded.
- Select one of the following:
  - WhenDiscovered - The driver is loaded when a business card is discovered (default value)
  - WhenMatched - The driver is loaded when a matching device name is discovered
  - UponConnection - The driver is loaded when another application requires it
- Select Save to save your selections and exit the panel.
- Select Cancel to exit without saving your selections.

```
com/ibm/joscfg/services/JSLBootProps.class=Service Installation Information
com/ibm/joscfg/services/JSLSimpleInstance.class=Service Definitions
com/ibm/joscfg/services/JSLDevProps.class=System Hardware Settings
com/ibm/joscfg/services/JSLLoadProps.class=Service Load Time
```

service.properties

The entries defined in service.properties are used by the JSL to load the driver dynamically. These include:

- `JSLBootProps` bean properties

```
bundleFileName=PC3Com.jar
bundleURL=file:/ROM/boot/
MCFBundleURL=file:/export/services/
bundleType=JAR
bundleInitClassName=sun.javaos.PC3Com.PC3Com
```

Note: The class within the driver that implements `serviceInstance` must be a subclass of `Service.`

- `JSLSimpleInstance` bean properties

```
bundleInstanceInfo=EthernetDriver,sun.javaos.PC3Com.PC3Com,
    sun.javaos.net.LanDriver,Ethernet
```

- `JSLLoadProps` bean properties

```
loadsUponConnection=false
loadsWhenMatchedToEntry=true
loadsWhenDiscovered=false
```

- `JSLDevProps` bean properties

```
matchingName=pci10b7,9050
compatibleMatchNames=pci10b7,5900 pci10b7,5950
```

Note: Because PXE is not implemented in Version 2 of JavaOS for Business, the `service.properties` sample code for the PC3Com driver contains hard-coded instructions for driver configuration. PXE allows the client to request that service properties for the driver be downloaded from the server.

Short descriptions of each `service.properties` attribute follow.

`bundleFileName`

The name of the bundle.

`bundleURL`

URL formatted String that contains the location of the bundled service.

`MCFBundleURL`

URL formatted String that contains the location of the Master Configuration File.

`bundleType`

A string specify the bundle packaging type. This parameter is used to match a service with an unbundler.

bundleInitClassName

Name of the class within the package that implements the JavaOS for Business `serviceInstance`. If this parameter is not present, static initialization of the package is not performed.

bundleInstanceInfo

A string contains the information necessary to advertise and create a Service instance.

loadsUponConnection

This service is only loaded when an application requests a ServiceConnection to the service.

loadsWhenMatchedToEntry

During the Device namespace manager discovery phase, the matchingName and compatibleMatchNames parameters are used to search for a matching service.

loadsWhenDiscovered

During the load phase, the Device namespace manager will run through all of the BusinessCard entries. If an entry is found with this property set, the service will be loaded.

matchingName

A string representing a unique device identification. This filed must follow the formatting rules specified in IEEE 1275.

compatibleMatchNames

A space delimited string containing compatible service matching names that may also be supported by this driver.

4

Driver Operations

Once a driver package has been unbundled and the driver classes loaded into the JavaOS for Business for Business system, driver operations may begin. In this chapter, a structured walkthrough of `PC3Com.java` sample code is used to illustrate common network driver operations.

The `PC3Com.java` code is presented here in its entirety and organized into eight segments. Each segment is prefaced with a narrative overview of the operations performed.

The purpose of this chapter is to present driver operations from two corollary points of view: one strategic (high-level narrative); the other tactical (low-level code). The narrative, in effect, describes the forrest. The sample code defines the trees.

PC3Com Sample Driver

The PC3Com class is a subclass of the `NetworkDriver` class, which is a subclass of the `DeviceDriver` class. `PC3Com.java` consequently inherits methods from each of these super classes.

To isolate functional components of the PC3Com class, the sample code is organized into the following segments:

- Class definition and interface implementations
- `ServiceInstance` methods
- Constructor function
- `DeviceDriver` methods
- `NetworkDeviceDriver` methods
- `EthernetDriver` methods

- Implementation dependent methods
- Constants

Interface Implementations

The PC3Com driver extends `NetworkDeviceDriver` and implements the following interfaces:

- `EthernetDevice`
- `ServiceInstance`

The EthernetDevice and ServiceInstance interfaces provide methods that the JSL uses to load and instantiate a driver. The `PC3Com` driver uses the interfaces to filter network broadcasts and perform peek and poke.

Class Extension Code -- Segment 1

```
package sun.javaos.PC3Com;

import java.io.IOException;

import javaos.javax.system.memory.*;
import javaos.javax.system.interrupts.*;
import javaos.javax.system.services.*;
import javaos.javax.system.database.*;
import javaos.javax.system.jdi.*;

//sun.javaos.* are not JDI compliant.
//It will be modified in the future.
import sun.javaos.net.*;
import sun.javaos.Debug;
import sun.javaos.Env;
import sun.javaos.BootConfiguration;

public final class PC3Com extends NetworkDeviceDriver
    implements EthernetDevice,EthernetDriver,ServiceInstance {

    private static Entry businessCard=null;
    private static PC3Com pc3Com=null;
```

Service Methods

As discussed in chapter 3, the Java Service Loader (JSL) consumes JSD events and dynamically loads services . The `Service` methods provided here allow the JSL to: `start`(), `stop`(), `suspend`(), and `resume`() the driver.

In the `start`() function, the JSL performs one-time driver initialization.

The JSL also calls `createInstance`() to instantiate a driver and `deleteInstance`() to remove a driver instance. In addition, it calls `getServiceParent`() to obtaina bus manager instance to pair with the driver.

`Service` variables are defined for the business card instance (which provides device configuration properties) and for the PC3Com driver instance (`pc3com`).

Service Code -- Segment 2

```
/**
     * Called by the JSL to do one time driver initialization.
     *
     * @param        businessCard the configuration properties of
     *               this device
     */
    public static void start(Entry businessCard) throws ServiceException {
        System.err.println("PC3Comstart, businessCard="+businessCard);
        businessCard=businessCard;
    }

    /**
     * Called by the JSL to stop this driver.
     */
    public static void stop() {}

    /**
     * Called by the JSL to suspend this driver.
     */
    public static void suspend() throws ServiceException {}
```

```
public Object getServingParent (Entry de) {
return null;
    }



//-------------------------------------------------------------
    // Implementation of ServiceInstance methods.
    // There are createInstance() and deleteInstance().
    //-------------------------------------------------------------

    /**
     * Called by JSL to request of a instance of a sevice.
     *
     * @param       businessCard the entry contains the configuration
     *              properties of this device
     * @param       logicalName the logical name of this device
     * @param       parent the parent ServiceInstance
     * @param       aliasEntry the alias of this device
     * @param       cookie generated by JSL, the driver can use
     *              this as an instanceID
     *
     * @return      the reference to this create instance
     *
     * @exception   ServiceException the service create error
     */
    public static ServiceInstance createInstance(
Entry businessCard, String logicalName, Object parent,
Entry aliasEntry, Object cookie) throws ServiceException {
```

```
System.err.println("PC3Com::createInstance, businessCard="
            +businessCard);
         System.err.println("PC3Com::createInstance, logicalName="
            +logicalName);
         System.err.println("PC3Com::createInstance, parent="
            +parent);
         System.err.println("PC3Com::createInstance, aliasEntry="
            +aliasEntry);
         System.err.println("PC3Com::createInstance, cookie="+cookie);

    // This implementation actually creates a new instance of the driver.
    if (pc3Com == null) {
            try {
        pc3Com=new PC3Com((Bus)parent, businessCard);
        System.err.println("PC3Com::createInstance, "
            + "binding PC3Com instance into env, tk="+pc3Com);
        // This is not JDI compliant.
        // It will be modified in the future.
        Env.bind("/dev/ether", pc3Com);
        } catch (Exception tex) {
                System.err.println("PC3Com::createInstance, "
            + "PC3Com() threw exception="+tex);
        throw new ServiceException(businessCard,tex.toString());
        }
    } else {
        System.err.println("PC3Com::createInstance, "
        + "returning reference to existing pc3com="+pc3Com);
    }
    return pc3Com;
     }

/**
     * Finalize this driver instance
     *
     * @param       cookie the instanceID
     *
     * @exception   ServiceException delete instance fail
     */
    public void deleteInstance(Object cookie) throws ServiceException {
    System.err.println("PC3Com::deleteInstance, PC3Com="+this);
    }
```

Constructor Function

The PC3Com constructor takes two parameters:

`Bus b` -- the bus manager for the device. In this case, the bus manager is the expansion bus manager for the PCI bus.

`Entry entry` -- the business card device entry in the JSD.

The constructor has two primary responsibilities:

- enable DMA by passing `isDMA,` a boolean property in the business card.
- `inititialize`() the instance with an interrupt source object, provided by the bus manager, which implements `connectInterruptSource` on behalf of the driver.

Constructor Code -- Segment 3

```
    /**
     * Constructor.
     *
     * @param       b driver's bus manager
     * @param       entry driver's businesscard
     *
     * @Exception   IOException memory allocation or interrupt
*               source connection failure
     */
    public PC3Com (Bus b, Entry entry) throws IOException {
super(entry, b);
bus = (ExpansionBus)b;

// Check the configuration information from the business card.
try {
    Object propValue = entry.getPropertyValue("isDma");
    if (propValue.toString().compareTo("true") == 0) {
PC3COMDMA = true;
    }
```

```
    } catch (Exception ignore) {
        println("PC3Com:catch " + ignore);
    }

    try {
        initialize();
        PC3ComInterruptSource pC3ComIntSrc =
        new PC3ComInterruptSource("PC3Com-0", this);
        bus.connectInterruptSource(pC3ComIntSrc);
    } catch (IOException e) {
        System.err.println("PC3Com constructor:  "
        + "initialization failed: " + e.getMessage());
        throw e;
    }
    }
```

`DeviceDriver` methods

Implementation of `DeviceDriver` methods. Only two:

- `openDriver()`
- `closeDriver()`

DeviceDriver Code -- Segment 4

```
/**
     * Called by the handle to open the Driver Instance
     *
     * @exception    DeviceException open device failure
     */
    protected void openDriver() throws DeviceException {
    return;
    }

/**
     * Called by the handle to close the Driver Instance
     *
     * @exception    DeviceException close device failure
     */
    protected void closeDriver() throws DeviceException {
    return;
    }
```

NetworkDeviceDriver methods

Implementation of NetworkDeviceDriver methods, including:

- getTransmitPacket()
- output()
- getConfig()
- report()
- getMtu()
- headerHint()
- getLocalAddress(),
- interfaceDescription()
- getStatistic()
- getHwType()
- enable()

NetworkDeviceDriver Code -- Segment 5

```
/**
     * Called by network layer to get a device specific packet.
     *
     * @param       hlen The size of the header
     * @param       dlen The size of the data
     *
     * @return      A GenericPacket.
     */
    public Packet getTransmitPacket(int hlen,int dlen) {
//return a GenericPacket with hlen+dlen+DWORD_PAD sized buffer.
//This is because the Programmed IO based output() rtn requires
//outputting to the nearest double word boundary past hlen+dlen,
//in double word chunks. The contents of the padded bytes are
//ignored.
Packet pkt = GenericPacket.get(hlen, dlen+DWORD_PAD);
pkt.setLength(hlen+dlen);
return pkt;
    }

/**
```

```
* Sends an IP packet out the network.
     *
     * @param        np the packet to be sent
     * @param        type type of packet
     * @param        destAddr the destination address
     */
    public void output(Packet np, int type, long destAddr) {
int i,j,len;
short status;

np.shiftHeader(-etherHeaderSize);
len = np.dataLength();

synchronized(this) {
    // Error Checking Code.
    if ((len > ALLOW_LARGE_PACKETS) || (len < etherHeaderSize)) {
ifOutDiscards++;
throw new Error(
    "PC3Com:output() - Bad Ethernet Packet size=" + len);
    }

    //Check if free space is availabel in FIFO. If not dump the packet.
    if ( (int)(ioaddr.getShort(TX_FREE) & 0xffff) < (len + 3) ) {
ifOutDiscards++;
return;
    }

    // put out the doubleword header
    ioaddr.setShort(TX_FIFO,(short)(len));
    ioaddr.setShort(TX_FIFO,(short)(0x00));

    np.putEthAddr(destAddr,0);
    np.putEthAddr(ourAddress,6);
    np.putShort(type,12);
```

```
    byte[] buf = ((GenericPacket)np).getByteArray();
    //buf was created with dword padding already
    int tmplen = (len + 3) & ~0x3;

    ioaddr.setIntArray(TX_FIFO,buf,0,(tmplen/4));

    ifOutOctets++;

        } //synchronized(this)
    }

    /**
     * To return default boot configuration parameters.
     * Since the ProtocolStack runs DHCP over this kind of link,
     * Ethernet, it will never ask us, so return null
     */
public BootConfiguration getConfig() {
return null;
    }

    /**
     * Print a summary of status.
     *
     * @param       out a PrintStream to use
     */
    public void report(java.io.PrintStream out) {
update_status();
out.print("PC3Com ethernet. Address = "
    + EthernetNetworkDriver.addrToString(ourAddress));
out.println("");
out.println("Received " + receiveCount + " packets with "
    + receiveErrors + " errors.");
out.println("Receive was busy " + rx_busy + " times.");

if(rx_over_errors > 0) {
    out.println("    " + rx_over_errors + " receive over errors.");
```

```
}
if(rx_length_errors > 0) {
    out.println("    " + rx_length_errors + " receive length errors.");
}
if(rx_frame_errors > 0) {
    out.println("    " + rx_frame_errors + " receive frame errors.");
}
if(rx_crc_errors > 0) {
    out.println("    " + rx_crc_errors + " receive crc errors.");
}

out.println("Transmitted " + tx_packets + " packets.");

if(tx_aborted_errors > 0) {
    out.println("    " + tx_aborted_errors +
" transmit aborted errors.");
}
if(tx_carrier_errors > 0) {
    out.println("    " + tx_carrier_errors +
" transmit carrier errors.");
}
if(tx_heartbeat_errors > 0) {
    out.println("    " + tx_heartbeat_errors +
" transmit heartbeat errors.");
}
if(collisions > 0) {
    out.println("    " + collisions + " collisions.");
}
if(tx_window_errors > 0) {
    out.println("    " + tx_window_errors +
" transmit window errors.");
}
if(tx_fifo_errors > 0) {
    out.println("    " + tx_fifo_errors + " transmit fifo errors.");
}
    }
```

```
/**
     * Our largest packet size
     *
     * @retrun      the maximum transmission unit
     */
    public int getMtu() {
return MTU;
    }

    /**
     * How many bytes of header our protocol layer and below requires
     *
     * @return      the length of header
     */
    public int headerHint() {
return HEADER_LENGTH;
    }

    /**
     * The ethernet address
     *
     * @return      the ether address
     */
    public long getLocalAddress() {
return ourAddress;
    }

/**
     * Description of ethernet
     *
     * @return      a String of interface description
     */
    public String interfaceDescription() {
return "10Mbps Ethernet interface";
    }
```

```
/**
     * Provide ethernet statistic information
     *
     * @param       index an index to statistic information
     * @return      value of a specific statistic information
     */
public int getStatistic(int index) {

        switch(index) {
        case 1:
            return 1;
        case 3:
            return 6;
        case 4:
            return 1500;
        case 5:
            return 100000000;
        case 7:
            return 1;
        case 8:
            return 1; // protocol == null ? 2 : 1;
        case 9:
            return 0;
        case 10:
            return ifInOctets;
        case 11:
            return ifInUcastPkts;
        case 12:
            return ifInNUcastPkts;
        case 13:
            return ifInDiscards;
        case 14:
            return ifInErrors;
        case 15:
            return ifInUnknownProtos;
case 16:
            return ifOutOctets;
        case 17:
            return ifOutUcastPkts;
        case 18:
            return ifOutNUcastPkts;
        case 19:
            return ifOutDiscards;
        case 20:
            return ifOutErrors;
        case 21:
            return ifOutQLen;
        }
```

```
 }
return -1;
    }


/**
     * Returns Arp type (Ethernet vs 802)
     */
    public int getHwType() {
      return 1;
    }
    /**
     *  EthernetNetworkDriver calls this to register an upcall object
     */
    public void enable(NetworkDriver net) {
    this.net = (EthernetNetworkDriver)net;
    }
```

`EthernetDriver` Methods

Implementation of `EthernetDriver` methods, including:

- `enablePromiscuous()`
- `disablePromiscuous()`
- `addMulticast(long addr)`
- `deleteMulticast(long addr)`

EthernetDriver Code -- Segment 6

```
//-------------------------------------------------------------
// Implementation of EthernetDriver methods
//-------------------------------------------------------------

/**
 * Turns on promiscuous mode
 */
public void enablePromiscuous() {
promiscuousFlag = RxProm;
updateRxControl();
}

/**
 * Turns off promiscuous mode
 */
public void disablePromiscuous() {
promiscuousFlag = 0;
updateRxControl();
}

/**
* Add a multicast address to listen to.
* This chip is stupid and does no filtering at all on multicasts.
* So, all we do is keep a use count of the number addresses added
* and subtracted. We count on our client never adding the same address
* twice, nor deleting an address that was never added.
*
* @param       addr Multicast address
*/
public void addMulticast(long addr) {
if (multicastCnt++ == 0)        // If previously off, then
    updateRxControl();          // turn 'em on.
}

/**
 * Delete a multicast address to listen to.
 * This chip is stupid and does no filtering at all on multicasts.
 * So, all we do is keep a use count of the number addresses added
 * and subtracted. We count on our client never adding the same address
 * twice, nor deleting an address that was never added.
 *
 * @param       addr Multicast address
 */
public void deleteMulticast(long addr) {
if (--multicastCnt == 0)        // If last address, then
    updateRxControl();          // turn off multicasts
}
```

EthernetDriver Code -- Segment 6

```
    //-------------------------------------------------------------
    // Implementation of EthernetDriver methods
    //-------------------------------------------------------------

    /**
     * Turns on promiscuous mode
     */
    public void enablePromiscuous() {
promiscuousFlag = RxProm;
updateRxControl();
    }

    /**
     * Turns off promiscuous mode
     */
    public void disablePromiscuous() {
promiscuousFlag = 0;
updateRxControl();
}

    /**
    * Add a multicast address to listen to.
    * This chip is stupid and does no filtering at all on multicasts.
    * So, all we do is keep a use count of the number addresses added
    * and subtracted. We count on our client never adding the same address
    * twice, nor deleting an address that was never added.
    *
    * @param       addr Multicast address
    */
    public void addMulticast(long addr) {
if (multicastCnt++ == 0)        // If previously off, then
    updateRxControl();          // turn 'em on.
    }
```

```
/**
     * Delete a multicast address to listen to.
     * This chip is stupid and does no filtering at all on multicasts.
     * So, all we do is keep a use count of the number addresses added
    * and subtracted. We count on our client never adding the same address
     * twice, nor deleting an address that was never added.
     *
     * @param       addr Multicast address
     */
    public void deleteMulticast(long addr) {
    if (--multicastCnt == 0)        // If last address, then
        updateRxControl();          // turn off multicasts
    }
```

Implementation Dependent Methods

The remaining methods of the PC3Com.java code deal with platform-specifc driver operations and are lumped together here out of convenience, rather than necessity. Although specific methods vary by processor, these routines nonetheless represent a set of common functions that driver writers must be aware of. The following low-level methods are presented in sample code:

- getIrq()
- boolean is3C905()
- config()
- busywait(int msec)
- readEEPROM(AccessibleMemory ioaddr, int index)
- reset()
- initalize()
- EL3WINDOW(int win_num)
- getStatus()
- returnTx(Packet pkt)
- printLatest()
- printPacket(Packet np,int len)
- printPacket(byte[] buf,int len)
- processInput()
- processInputDMA()
- update_status()
- updateRxControl()
- println(String s)
- print(String s)

Dependent Code -- Segment 7

getIrq()

```
/**
     * Bus Manager calls this method to probe if it requires a
     * hardcoded irq. If yes, return a non-negative number. If no,
     * return -1.
     *
     * @return interrupt request level
     */
    public int getIrq() {
    return irq;
    }
```

config()

```
private void config() throws IOException {
int i,selectedIOAddr;

System.err.println("PC3Com start config()");

MemoryDescriptor[] mDescs;

try {
    constraints = null;

    /*
```

```
     * Obtain PCI memory descriptors from bus manager.
     */
    mDescs = bus.getMemoryDescriptors();
    System.out.println("----- PC3Com -----");
    System.out.println(mDescs.length + " memory descriptors");
    for (int x = 0; x < mDescs.length; x++) {
PCIDeviceInfo info = (PCIDeviceInfo)mDescs[x].getDevInfo();
System.out.println(x + ": '" + info.getName() +
"' (" + mDescs[x] + ")");
    }

    /*
    * Looke for the PCI I/O memory descriptor and allocate accessible
    * memory from it.
    *
    * Can use the devInfo name, or instead use its getSpace() method
    * and check that way.
    */
    int mdIdx;
    for (mdIdx = 0; mdIdx < mDescs.length; mdIdx++) {
PCIDeviceInfo devInfo =
    (PCIDeviceInfo)mDescs[mdIdx].getDevInfo();
if (devInfo.getName().equals(PCIBusManager.PCI_IONAME)) {
    constraints = bus.allocGenericMC(PCIBusManager.PCI_IONAME);
    constraints.setAllocAddrs(new Address32(0, 16),
new Address32(0xFFFF, 16));
    System.out.println("PC3Com: io index is " + mdIdx);
    break;
}
    }
    ioaddr = bus.allocAccessibleMemory(mdIdx, constraints);

} catch (InvalidAddressException e) {
    System.err.println(e);
    // For now ignore it.
```

```
} catch (AllocationException e) {
    System.err.println(e);
    // For now ignore it.
}


// Get IRQ from bus manager
irq = ((PCIBusManager)bus).getIntLine();

// Figure out our hardware ethernet address.
EL3WINDOW(0);
ourAddress = 0;
for(i=0;i<3;i++) {
    short rp;
    rp = readEEPROM(ioaddr,i);
    println("readEEPROM = 0x"+Integer.toHexString(rp));
    ourAddress = (ourAddress<<16) | ((long)rp & 0xFFFFL);
}

// The if_port symbol can be set when the module is loaded.
if_port = (short)(readEEPROM(ioaddr, 8) >> 14);

// This is vital: the transceiver used must be set in the resource
//  configuration register.  It took me many hours to discover this.
ioaddr.setShort(6, (short)(if_port << 14));

// Get the DeviceId
deviceId = (int)(readEEPROM(ioaddr, 3) & 0xffff);
switch (deviceId) {
case 0x9050:
    System.err.println("Fast EtherLink XL PCI 10/100 Mbps" +
    ", Shared 10BASE-T/100BASE-TX");
    break;
case 0x5950:
    System.err.println("Fast EtherLink PCI" +
    ", Shared 10BASE-T/100BASE-TX");
    break;
case 0x5900:
    System.err.println("EtherLink III PCI" +
    ", 10Mbps");
    break;
}
println("PC3Com deviceId = 0x" + Integer.toHexString(deviceId));

System.err.println("PC3Com 3c590/90x, "
    + ioaddr.toString() + ", IRQ = 0x"
    + Integer.toHexString(irq));
System.err.println("Ether address = "
    + EthernetNetworkDriver.addrToString(ourAddress));
}
```

busywait(int msec)

```
private void busywait(int msec) {
int i,j,s=0;
for(i=0;i<msec;i++) {
    for(j=0;j<1000;j++) {
    s++;
    }
}
}
```

readEEPROM(AccessibleMemory ioaddr, int index)

```
private short readEEPROM(AccessibleMemory ioaddr, int index) {
ioaddr.setShort(10, (short)(EEPROM_READ + index));
// Pause for at least 162 us. for the read to take place.
busywait(162);
return ioaddr.getShort((short)12);
}
```

reset()

```
// Reset and restore all of the 3c590 registers.
private void reset() {
int i;
println("reset now.");

EL3WINDOW(0);

ioaddr.setShort(4,(short)0x0001);    /* Activate board. */
ioaddr.setShort(6,(short)(if_port<<14));/* set the xcvr. */
ioaddr.setShort(8,(short)0x3f00);    /* Set the IRQ line. */

/* Set the station address in window 2. */
```

```
EL3WINDOW(2);
ioaddr.setByte(0,(byte)((ourAddress >> 40) & 0xFF));
ioaddr.setByte(1,(byte)((ourAddress >> 32) & 0xFF));
ioaddr.setByte(2,(byte)((ourAddress >> 24) & 0xFF));
ioaddr.setByte(3,(byte)((ourAddress >> 16) & 0xFF));
ioaddr.setByte(4,(byte)((ourAddress >> 8) & 0xFF));
ioaddr.setByte(5,(byte)((ourAddress) & 0xFF));

if (if_port == 3) {
    // Start the thinnet transceiver. We should really wait 50ms...
    ioaddr.setShort(EL3_CMD, StartCoax);
} else if (if_port == 0) {
    // 10baseT interface, enabled link beat and jabber check.
    EL3WINDOW(4);
    ioaddr.setShort(WN4_MEDIA,
    (short)(ioaddr.getShort(WN4_MEDIA)|MEDIA_TP));
}

// Switch to the stats window, and clear all stats by reading.
ioaddr.setShort(EL3_CMD,StatsDisable);

EL3WINDOW(6);
for (i = 0; i < 9; i++) {
    ioaddr.getByte(i);
}
ioaddr.getShort(10);
ioaddr.getShort(12);

/* Switch to register set 1 for normal use. */
EL3WINDOW(1);

/* Accept b-cast and phys addr only. */
ioaddr.setShort(EL3_CMD, (short)(SetRxFilter|RxStation|RxBroadcast));
// Turn on statistics.
//ioaddr.setShort(EL3_CMD, StatsEnable);
ioaddr.setShort(EL3_CMD, RxEnable); // Enable the receiver.
ioaddr.setShort(EL3_CMD, TxEnable); // Enable transmitter.
synchronized (signal) {
    /* Allow status bits to be seen. */
    ioaddr.setShort(EL3_CMD, (short)(IndicationEnable | 0xff));
    ioaddr.setShort(EL3_CMD, (short)(AckIntr | 0x69)); // Ack IRQ
    //ioaddr.setShort(EL3_CMD, (short)(SetIntrMask | 0x90));
    ioaddr.setShort(EL3_CMD, (short)(SetIntrMask | 0x10));
    // Set interrupt mask.
}
}
```

initalize()

```
private void initialize() throws IOException {

signal = new Object();

try {
    config();
} catch (IOException e) {
    throw e;
}

if (PC3COMDMA) {
    println("PC3Com does DMA");

    // Allocate the DMA memory and map to AccessibleMemory
    try {
MemoryConstraints dmaMemCons = bus.allocGenericMC("dma");

dmaMemory = bus.allocDMAMemory(
    new Address32(largestRx*rxRingSize, 16), dmaMemCons);

MemoryConstraints accMemCons =
    bus.allocGenericMC(PCIBusManager.PCI_MEMNAME);

accMemCons.setCacheMode(MainMemory.CACHE_MODE_DEFAULT);
dmaAccMemory = bus.allocAccessibleMemory(dmaMemory, accMemCons);

    } catch (Exception ex) {
System.err.println(
    "Caught Exception while allocating dmaMemory: " + ex);
ex.printStackTrace();
    }
```

```
    // Construct packets
    Memory subRangeDmaMemory = null;
    AccessibleMemory subRangeDmaAccMemory = null;
         rxDmaPacket = new PC3ComDmaPacket[rxRingSize];

         for (int i = 0; i < rxRingSize; ++i) {

    try {
        subRangeDmaMemory = bus.getSubRangeDMA(
        dmaMemory,
        new Address32(largestRx*i) ,
        new Address32(largestRx));

        subRangeDmaAccMemory = bus.getSubRangeAM(
        dmaAccMemory,
        new Address32(largestRx*i) ,
        new Address32(largestRx));

        rxDmaPacket[i] = new PC3ComDmaPacket(
        bus, subRangeDmaMemory, subRangeDmaAccMemory);
    } catch (InvalidAddressException e) {
        System.err.println("Caught Exception: " + e);
    } catch (AllocationException e) {
        System.err.println("Caught Exception: " + e);
    }
    }

} else {
    println("PC3Com does PIO");
    rxPacket = new PC3ComPacket[rxRingSize];
    for (int i = 0; i < rxRingSize; ++i) {
    rxPacket[i] = new PC3ComPacket(largestRx);
    }
}

    // And start the chip up.
    reset();
}
```

EL3WINDOW(int win_num)

```
private void EL3WINDOW(int win_num) {
ioaddr.setShort(EL3_CMD,(short)(SelectWindow+win_num));
}
```

getStatus()

```
private int getStatus() {
return ioaddr.getShort(EL3_STATUS);
}
```

returnTx(Packet pkt)

```
public void returnTx(Packet pkt) {
}
```

printLatest()

```
public static void printLatest(){
int i;
for(i=0;i<latestLen;i++) {
    int v;
    v = (latestPacket[i] & 0xFF);
    if(v<0x10) {
    print("0"+Integer.toHexString(v));
    } else {
    print(Integer.toHexString(v));
    }
    if((i & 15)==15) {
    print("\n");
    } else {
    print(":");
    }
}
print("\n");
}
```

printPacket(Packet np,int len)

```
private void printPacket(Packet np,int len) {
int i,v;
if(!PC3Com.debug) {
    return;
}
for(i=0;i<len;i++) {
    v = (np.getByte(i) & 0xFF);
    if(v<0x10) {
    Debug.print("0"+Integer.toHexString(v));
    } else {
    Debug.print(Integer.toHexString(v));
    }
    if((i & 15)==15) {
    Debug.print("\n");
    } else {
    Debug.print(":");
    }
}
Debug.print("\n");
}
```

printPacket(byte[] buf,int len)

```
private void printPacket(byte[] buf,int len) {
int i,v;
if(!PC3Com.debug) {
    return;
}
for(i=0;i<len;i++) {
    v = (buf[i] & 0xFF);
    if(v<0x10) {
    Debug.print("0"+Integer.toHexString(v));
    } else {
    Debug.print(Integer.toHexString(v));
    }
    if((i & 15)==15) {
    Debug.print("\n");
    } else {
    Debug.print(":");
    }
}
Debug.print("\n");
}
```

The following information applies to the `handleInterrupt`() and `processInput`() routines, which employ *synchromization*.

The purpose of using *synchronization* in `PC3Com` is to ensure mutually exclusive access to resources when multiple threads compete for the same resource object. Resources include hardware registers, global fields, and packets.

The `output`() method is called by the network layer (IP, TFTP, ARP, etc.) to send data to the outside network. The `input`() method is called from the interrupt handle to move the data from the ehternet adapter's buffer to system memory.

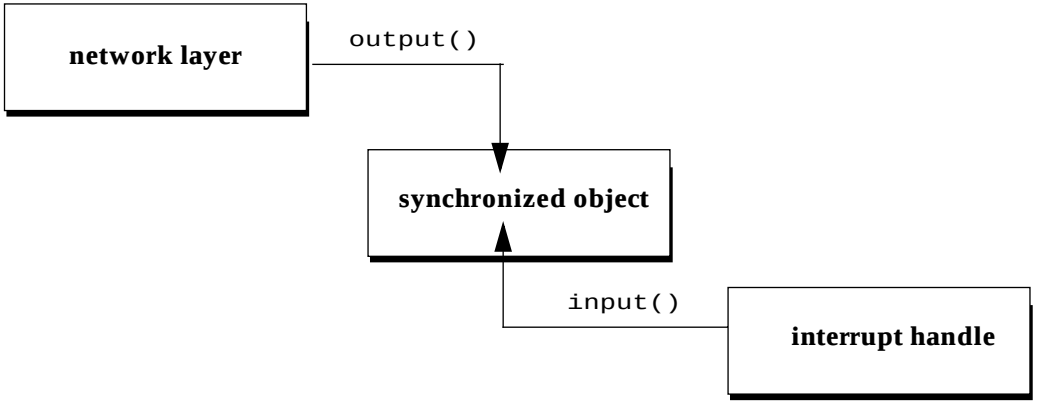

FIGURE 4-1 Synchronization

handleInterrupt()

```
public void handleInterrupt() {
short status;
synchronized (signal) {
    status = ioaddr.getShort(EL3_STATUS);
    if ((status & 0x11) != 0) {
    ioaddr.setShort(EL3_CMD,(short)(IndicationEnable | 0x00));
    try {
        int returncode=0;
        do {
        if (PC3COMDMA) {
            returncode = processInputDMA();
        } else {
            returncode = processInput();
        }
        //>0 means possible more processing
        } while (returncode > 0);
    } catch (InterruptedException e) {
        System.out.println("handleInterrupt caught " + e);
        e.printStackTrace();
    }
    // Acknowledge the IRQ.
    ioaddr.setShort(EL3_CMD, (short)(AckIntr|0x01));
    // Resetting the IndicationEnable to 0xff done by input rtn.
    }
}
}
```

processInput()

```
    private int processInput() throws InterruptedException {
int i,j,len;
short status;
PC3ComPacket packet;

synchronized (this) {

    status = ioaddr.getShort(RX_STATUS);
    packet = null;

    if (status > 0) {
if((status & RXERROR) == RXERROR) {    // Error. Update Status.
    short error = (short)(status & 0x3800);
    ifInErrors++;
    receiveErrors ++;
    switch (error) {
case RXERROR_OVERRUN:
    rx_over_errors++;
    break;
case RXERROR_RUNTFRAME:
    rx_length_errors++;
    break;
case RXERROR_ALIGNMENT:
    rx_frame_errors++;
    break;
case RXERROR_RUNTFRAME|RXERROR_ALIGNMENT:
    rx_length_errors++;
    break;
case RXERROR_OVERSIZEDFRAME:
    rx_frame_errors++;
    break;
case RXERROR_CRC:
    rx_crc_errors++;
    break;
    }
```

```
    ioaddr.setShort(EL3_CMD, RxDiscard); // Rx discard
    return 1;
}

len = status & 0x7ff;
int tmplen = ((len + 3) & ~3);
packet = rxPacket[rxIndex];

if (packet.packetInUse()) {
    ifInDiscards++;
    rx_busy++;
    ioaddr.setShort(EL3_CMD, RxDiscard);
    return 1;
}

packet.init(0, 0, len);
byte[] buf = packet.getByteArray();
ioaddr.getIntArray(RX_FIFO,buf,0,(tmplen/4));

// Pop top Rx packet
ioaddr.setShort(EL3_CMD,RxDiscard);
ifInOctets++;

    } //if (status>0)

} //synchronized(this)

if (status > 0) {
    try {
if (net != null) {
    // Deliver the packet to higher.
    net.input((Packet)packet);
}
    } finally {
packet.recycle();
    }
```

```
                    rxIndex = (rxIndex + 1) % rxRingSize;
                    receiveCount++;
                    return 1;
                } else {
                    // Reenable indications of interrupts
                    ioaddr.setShort(EL3_CMD,(short)(IndicationEnable | 0xff));
                    return 0;
                }
                }
```

processInputDMA()

```
    private int processInputDMA() throws InterruptedException {
int len;
short status;
PC3ComDmaPacket packet;

        synchronized (this) {

            status = ioaddr.getShort(RX_STATUS);
            packet = null;

            if (status > 0) {

                if((status & RXERROR) == RXERROR) {    // Error. Update Status.
                    short error = (short)(status & 0x3800);
                    ifInErrors++;
                    receiveErrors ++;
    switch (error) {
case RXERROR_OVERRUN:
    rx_over_errors++;
    break;
case RXERROR_RUNTFRAME:
    rx_length_errors++;
    break;
```

```
case RXERROR_ALIGNMENT:
    rx_frame_errors++;
    break;
case RXERROR_RUNTFRAME|RXERROR_ALIGNMENT:
    rx_length_errors++;
    break;
case RXERROR_OVERSIZEDFRAME:
    rx_frame_errors++;
    break;
case RXERROR_CRC:
    rx_crc_errors++;
    break;
    }
                    ioaddr.setShort(EL3_CMD, RxDiscard); // Rx discard
                    return 1;
                }

                len = status & 0x7ff;
                packet = rxDmaPacket[rxIndex];

                if (packet.packetInUse()) {
                    rx_busy++;
                    ifInDiscards++;
                    ioaddr.setShort(EL3_CMD,RxDiscard);
                    return 1;
                }

                packet.init(0, 0, len);
```

```
EL3WINDOW(7);

                //Set up the Upload registers for DMA and start DMA
                ioaddr.setInt(MASTER_ADDRESS,
      packet.getMemBaseAddr().intValue());
                ioaddr.setShort(MASTER_LEN, (short)len);
                ioaddr.setShort(EL3_CMD, (short)(StartDma | Upload));
                //Poll the MASTER STATUS Register for completion
                int loop_detect;
                short uploadStatus;

loop_detect = 100;
                while (--loop_detect > 0) {
                    uploadStatus = ioaddr.getShort(MASTER_STATUS);
                    if ((uploadStatus & masterUpload) != 0) {
                        ioaddr.setShort(MASTER_STATUS, masterUpload);
                        break;
                    }
                }

EL3WINDOW(1);

                // Pop top Rx packet
                ioaddr.setShort(EL3_CMD,RxDiscard);

                if (loop_detect <= 0) {
                    ifInErrors++;
                    packet.recycle();
                    return 1;
                }

                ifInOctets++;
            }

} //synchronized(this)
```

```
if (status > 0) {
    try {
    if (net != null) {
        // Deliver the packet to higher.
        net.input((Packet)packet);
    }
    } finally {
    packet.recycle();
    }

    rxIndex = (rxIndex + 1) % rxRingSize;
    receiveCount++;
    return 1;
} else {
    // Reenable indications of interrupts
    ioaddr.setShort(EL3_CMD,(short)(IndicationEnable | 0xff));
    return 0;
    }
}
```

etDebug(boolean enable)

```
// setDebug can be used to turn on or off debug error messages.
public static void setDebug(boolean enable) {
    debug = enable;
}
```

supdate_status()

```
private void update_status() {
    if (stats == false) {
        return;
    }
    // Turn off the statistics updates while reading.
    ioaddr.setShort(EL3_CMD,StatsDisable);
    // Switch to the stats window, and read everything.
    EL3WINDOW(6);
    tx_carrier_errors +=    ioaddr.getByte(0);
    tx_heartbeat_errors +=    ioaddr.getByte(1);
    /* multiple_collisions += */    ioaddr.getByte(2);
    collisions +=        ioaddr.getByte(3);
    tx_window_errors +=    ioaddr.getByte(4);
    rx_fifo_errors +=    ioaddr.getByte(5);
    tx_packets +=        ioaddr.getByte(6);
    /* rx packets += */    ioaddr.getByte(7);
    /* tx deferrals += */    ioaddr.getByte(8);
    ioaddr.getByte(9);
    /* total rx octets += */    ioaddr.getShort(10);
    /* total tx octets += */    ioaddr.getShort(12);

    // Back to window 1, and turn statistics back on.
    EL3WINDOW(1);
    ioaddr.setShort(EL3_CMD, StatsEnable);
    return;
}
```

updateRxControl()

```
/**
 * Rewrites the receive control register with current value.
 * Always enable station reception and broadcasts.
 * Logical OR in promiscuous bit and multicast bit if appropriate
 */
private void updateRxControl() {
ioaddr.setShort(EL3_CMD,
(short)(SetRxFilter | RxStation | RxBroadcast | promiscuousFlag |
    ((multicastCnt != 0) ? RxMulticast : 0)));
}
```

uprintln(String s)

```
    private static void println(String s) {
    if(PC3Com.debug) {
        Debug.println(s);
    }
    }
```

print(String s)

```
private static void print(String s) {
if(PC3Com.debug) {
    Debug.print(s);
}
     }
```

Constants -- Segment 8

```
ExpansionBus bus;   // Reference to the bus
private ProtocolStack protocol; // object to deliver input() upcalls to
private Object signal;
private int deviceId;

private long ourAddress;
private MemoryConstraints constraints;
private AccessibleMemory ioaddr;
private int receiveCount = 0;
private int receiveErrors = 0;
private int rx_over_errors = 0;
private int rx_length_errors = 0;
private int rx_frame_errors = 0;
private int rx_fifo_errors = 0;
private int rx_crc_errors = 0;
private int tx_count = 0;
```

```
private int tx_aborted_errors = 0;
    private int tx_carrier_errors = 0;
    private int tx_heartbeat_errors = 0;
    private int collisions = 0;
    private int tx_window_errors = 0;
    private int tx_fifo_errors = 0;
    private int tx_packets = 0;
    private int rx_busy = 0;
    private Memory dmaMemory;
    private AccessibleMemory dmaAccMemory;
    private PC3ComPacket rxPacket[];
    private PC3ComDmaPacket rxDmaPacket[];
    private int rxIndex = 0;
    private int promiscuousFlag;        // Set to RxProm if promiscuous
    private int multicastCnt;            // Cnt of # of multicasts added

    // constants of etherpacket header size
    private static final int etherHeaderSize = 14;
    private static final int rxRingSize = 32;
    private static final int largestRx = 1792;
    // max pad bits to pad to a dword boundary
    private static final int DWORD_PAD = 3;

// offsets from base I/O address.
    private static final int EL3_DATA    = 0x00;
    private static final int EL3_CMD    = 0x0e;
    private static final int EL3_STATUS = 0x0e;
    private static final int ID_PORT    = 0x100;
    private static final int EEPROM_READ = 0x80;
```

```
// The top five bits written to EL3_CMD are a command,
    // the lower 11 bits are the parameter, if applicable.
    private static final short TotalReset = (short)(0<<11);
    private static final short SelectWindow = (short)(1<<11);
    private static final short StartCoax = (short)(2<<11);
    private static final short RxDisable = (short)(3<<11);
    private static final short RxEnable = (short)(4<<11);
    private static final short RxReset = (short)(5<<11);
    private static final short TxDone = (short)(7<<11);
    private static final short RxDiscard = (short)(8<<11);
    private static final short TxEnable = (short)(9<<11);
    private static final short TxDisable = (short)(10<<11);
    private static final short TxReset = (short)(11<<11);
    private static final short FakeIntr = (short)(12<<11);
    private static final short AckIntr = (short)(13<<11);
    private static final short SetIntrMask = (short)(14<<11);
    private static final short IndicationEnable = (short)(15<<11);
    private static final short SetRxFilter = (short)(16<<11);
    private static final short SetRxThreshold = (short)(17<<11);
    private static final short SetTxThreshold = (short)(18<<11);
    private static final short SetTxStart = (short)(19<<11);
    private static final short StartDma = (short)(20<<11);
    private static final short StatsEnable = (short)(21<<11);
    private static final short StatsDisable = (short)(22<<11);
    private static final short StopCoax = (short)(23<<11);

// The SetRxFilter command accepts the following classes:
    private static final short RxStation = 1;
    private static final short RxMulticast = 2;
    private static final short RxBroadcast = 4;
    private static final short RxProm = 8;

    // The StartDma command accepts the following paramaters
    private static final short Upload = 0x00;
    private static final short Download = 0x01;
```

```
    // Register window 1 offsets, the window used in normal operation.
    private static final short TX_FIFO = 0x00;
    private static final short RX_FIFO = 0x00;
    private static final short RX_ERROR = 0x04;
    private static final short RX_STATUS = 0x08;
    private static final short TX_STATUS = 0x0B;
    //Remaining free bytes in Tx buffer.
    private static final short TX_FREE = 0x0C;
    //Window0: Set IRQ line in bits 12-15.
    private static final short WN0_IRQ = 0x08;
    //Window4: Various transcvr/media bits.
    private static final short WN4_MEDIA = 0x0A;
    //Enable link beat & jabber for 10baseT.
    private static final short MEDIA_TP = 0x00C0;

    // Register window 7 offsets, the default window
    private static final int   MASTER_ADDRESS = 0x00;
    private static final short MASTER_LEN = 0x06;
    private static final short MASTER_STATUS = 0x0C;

// Register window 7 bits
    private static final short masterUpload = (short)(1<<14);
    private static final short masterDownload = (short)(1<<12);

    // Frame Start Header Constants
    private static final short TX_INDICATE = (short)0x8000;

    private boolean stats = false;
    private short if_port;
```

```
private int irq;
    private static byte[] latestPacket;
    private static int latestLen;
    private static boolean debug = false;
    private GenericPacket inPacket;
    private boolean inPacketp;
    // object to deliver input() upcalls to.
    private EthernetNetworkDriver net;

// Statistic
    private static int ifOutQLen = 0;
    private static int ifOutErrors= 0;
    private static int ifOutDiscards = 0;
    private static int ifOutNUcastPkts = 0;
    private static int ifOutUcastPkts = 0;
    private static int ifOutOctets = 0;
    private static int ifInUnknownProtos = 0;
    private static int ifInErrors = 0;
    private static int ifInDiscards = 0;
    private static int ifInNUcastPkts = 0;
    private static int ifInUcastPkts = 0;
    private static int ifInOctets = 0;

// RxError
    private static final int RXERROR                = 0x4000;
    private static final int RXERROR_OVERRUN        = 0x0000;
    private static final int RXERROR_RUNTFRAME      = 0x0800;
    private static final int RXERROR_ALIGNMENT      = 0x1000;
    private static final int RXERROR_OVERSIZEDFRAME = 0x2000;
    private static final int RXERROR_CRC            = 0x2800;

    private static final int MTU = 1514;
    private static final int ALLOW_LARGE_PACKETS = 1536;
    private static final int HEADER_LENGTH = 14;

    private boolean PC3COMDMA = false;
}
```

5

Bus Manager Services

Bus Managers insulate device drivers from platform and bus dependencies by providing low-level system services such as memory allocation and interrupt processing. This allows drivers to be written entirely in Java and remain architecture neutral. It also simplifies the task of driver writing by removing bus management concerns. For driver writers, the result is streamlined access to system resources, reduced code overhead, and faster development cycles.

Memory Architecture

The JavaOS for Business memory architecture was designed to provide platform-independent classes for drivers to access memory. Regardless of the underlying platform, drivers always use the same class methods to read and write memory.

Methods are also provided for bus managers to construct and allocate objects, called *memory descriptors*, for later use by device drivers. These memory objects are generic in nature, but ultimately facilitate access to lower-level memory classes, such as `RegularMemory`, `IOMemory`, and `PortIOMemory`. Because bus managers are trusted code, they are granted low-level access to system resources and act as *memory object factories*, producing and distributing memory (and accompanying address objects) to drivers as needed.

To maintain driver portability and to safeguard system resources, device drivers are only allowed to directly access four `Memory` classes, arranged in two main hierarchies. These include:

- `Memory/DMAMemory`
- `Memory/MainMemory/AccessibleMemory`

It is important to note that the Memory/DMA and Memory/MainMemory/AccessibleMemory classes do not represent a subsystem implementation for memory management. In fact, the functionality for managing the various types of memory (virtual, physical, I/O, and DMA) is provided by the *microkernel*. The memory classes simply take advantage of pre-existent microkernel functionality, leveraging rather than reinventing the kernel.

The memory classes available for direct use by drivers (excluding shaded rectangles) are illustrated in the following diagram:

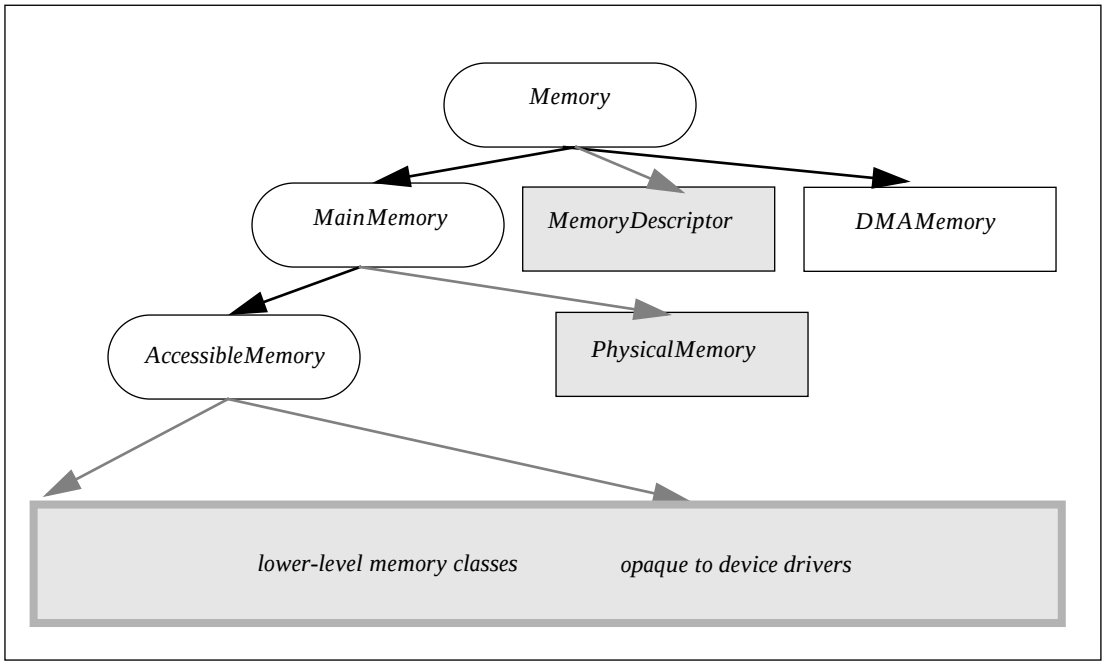

FIGURE 1-1 Driver memory classes.

Three distinct hierarchies are shown extending *Memory*:

- Memory/DMAMemory
- Memory/MemoryDescriptor
- Memory/MainMemory/AccessibleMemory

The MemoryDescriptor class and the PhysicalMemory class, which is a subclass of MainMemory, are used exclusively by bus managers.

Device drivers remain blissfully unaware of memory classes below DMAMemory and AccessibleMemory and have no notion of bus manager classes at any level of the memory hierarchy.

All memory objects are allocated via APIs of the driver's bus manager. These APIs are discussed later under `ExpansionBus`.

Security

Since driver and device matching information is already stored in the JSD, drivers are never required to specify which device they are managing. This provides added security to the system, since a driver cannot accidentally allocate memory intended for another device.

Classes and Interfaces

Following is a description of the memory classes illustrated earlier, as well as a discussion of memory constraints.

`Memory` Class

The `Memory` class, the highest level class in the memory hierarchy, is extended by `DMAMemory` and `MainMemory`. `Memory` is an abstract class, composed of such general attributes as a base address, length, and various memory constraints.

Under the `Memory` class, two subclasses are extended by device drivers:

- `MainMemory` -- memory accessed ultimately by the CPU, although perhaps not directly, as is the case with `PhysicalMemory`
- `DMAMemory` -- memory accessed by DMA external to the CPU

`MemoryDescriptor` Class

Having a non-abstract (and therefore instantiable) sub-class of `Memory` proves useful for representing fundamental characteristics of memory, such as base address and length, before an actual memory object has been constructed. Other than being instantiable, the `MemoryDescriptor` class does not provide any additional functionality beyond that provided by `Memory`. It is used by bus managers to allocate memory objects for driver use.

Because memory descriptors are not of any particular memory type, they can represent physical memory, DMA memory, virtual memory--*virtually* anything, up to and including a piece of device space on a bus.

When a driver is loaded by the JSL, the bus manager allocates an array of `MemoryDescriptor` objects for the device associated with the driver. Each descriptor specifies memory that the driver can use to map device registers, device interrupts, and so on.

Since `MemoryDescriptor` is a sub-class of `Memory`, the driver can call the `getMemBaseAddr`() method to get the base address of the memory, and the `getmemLength`() method to get the length. It can also call the `getAddressSpace`() method of `MemoryDescriptor` to get the particular address space managed by the bus manager that this memory pertains to. In many cases, these three pieces of information are sufficient to identify the memory.

The `MemoryDescriptor` method `getDevInfo`() returns an object of type `Object`, which is different for each particular bus manager. This *devinfo* object provides methods specific to the bus manager, which can be used to further identify the memory. In the case of PCI, for example, the object returned is of type `PCIDeviceInfo`.

`MainMemory` Class

The `MainMemory` class specifies abstract methods for managing *caching*, which is the only functionality common to its sub-classes.

Eventually in the hierarchy (in `PhysicalMemory`, `VirtualMemory`, and `PortIOMemory`, for example), the abstract cache management methods are implemented. Cache management is highly platform-dependent; consequently, each of these implementations ultimately relies on invoking the microkernel to flush caches, set the cache mode, and so on.

`MainMemory` is extended by:

- `AccessibleMemory`, which represents all memory that can be directly accessed by drivers.
- `PhysicalMemory`, which represents physical memory and may not be directly accessed by drivers.

PhysicalMemory Class

Physical memory is not available for use by device drivers. Only bus managers can access physical memory. This prevents drivers, which are non-trusted code, from accessing low-level system resources, preserving system integrity and isolating drivers from platform specifics.

DMAMemory Class

The DMAMemory class provides methods for setting up/tearing down DMA mappings to physical memory (ultimately done by the microkernel running below these classes).

To avoid reinventing functionality already supported by the microkernel, this class does *not* abstract common DMA hardware methods--a difficult task in any case given the abundant flavors of DMA hardware in the marketplace.

The DMAMemory object is not concerned with whether the DMA addresses that the microkernel sets up for it are virtual DMA addresses that map through some IOMMU to physical DMA addresses, or whether the DMA addresses are physical DMA addresses. To the memory object--and to the driver writer--the addresses are simply DMA addresses, and such mappings are transparent to DMAMemory, which is maintained entirely by the microkernel.

Memory is the superclass of DMAMemory. Because the DMAMemory class is not part of javax, a casted Memory object is returned to the driver when allocating DMA memory. Consequently, the only methods the driver can use are those of Memory.

Methods for allocating/deallocating DMA memory are described in the ExpansionBus section later in this chapter. Drivers obtain DMA memory from bus managers.

AccessibleMemory Class

AccessibleMemory is JavaOS for Business terminology for the accessible memory given to drivers by bus managers. Using the AccessibleMemory class, drivers are ensured portability across platforms. AccessibleMemory defines platform-independent abstract methods, such as setByte*()*. Drivers should only use:

- methods abstracted in AccessibleMemory
- methods abstracted or implemented by MainMemory and Memory, the superclasses of AccessibleMemory.

PhysicalMemory and DMAMemory classes--along with *all* classes below AccessibleMemory--should only be accessed by bus managers, whose job it is to isolate drivers from platform dependencies.

AccessibleMemory is extended by:

- VirtualMemory, if the memory is accessed by a CPU issuing normal load and store instructions.
- PortIOMemory, if the memory is accessed by a CPU issuing special I/O port instructions.

Methods for allocating/deallocating accessible memory are described in the ExpansionBus section later in this chapter.

MemoryConstraints Class

Before a driver can request memory allocation, it must specify a MemoryConstraints object. Memory constraints fall into two general categories:

- *allocation* constraints
- *access* constraints

The following code fragment defines the set of possible memory constraints:

```
public class MemoryConstraints {

    Address allocMinAddr;       // minimum valid address
    Address allocMaxAddr;       // maximum valid address
    int allocAlign;             // alignment bit to enforce for allocations
    int cacheMode;              // as defined in MainMemory.java.
    boolean locked;             // whether to be allocated locked.
    boolean waitForMem;         // wait for memory on allocate.
    boolean typeSwapped;        // byte-swapped type memory.
    int endian;                 // endianness.
    boolean typeIO;             // io (either Port or Virtual I/O).
    boolean typeVirtual;        // sub-class of VirtualMemory.
    int accessMinSize;          // minimum size of access (bytes).
    int accessMaxSize;          // maximum size of access (bytes).
    int accessAlign;            // alignment of access.
    int readwrite;              // READONLY (0) or READWRITE (1).
```

Bus drivers simplify the task of memory constraint specification for drivers by providing an interface, allocGenericMC(), which returns a default MemoryConstraints object for a particular address space that the bus manager controls. Instead of having to list the complete set of constraints, the driver need only specify a subset. Optionally, driver writers can ignore this method and instantiate a MemoryConstraints object, spelling out each constraint individually.

Numerous *get/set* methods manipulate constraint values.defined in the MemoryConstraints class.

ExpansionBus Sample Code

Device drivers use the methods defined in ExpansionBus to request the system resources discussed. ExpansionBus code is displayed and annotated below:

```
public interface ExpansionBus extends Bus {
    /**
     * Return an enumerator for the address spaces on this bus.
     */
    public Enumeration getAddressSpaces();

    /**
     * Return an address space with the given name.
     */
    public AddressSpace getAddressSpace(String spaceName);

    public MemoryConstraints allocGenericMC(String asName)
        throws InvalidAddressException;
```

Methods for mapping MemoryDescriptor *objects*

```
    public MemoryDescriptor[] getMemoryDescriptors();
```

Methods for allocating/freeing AccessibleMemory

```
    public AccessibleMemory allocAccessibleMemory(int mdIdx,
        MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public AccessibleMemory allocAccessibleMemory(MemoryDescriptor m,
        MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public AccessibleMemory allocAccessibleMemory(Address len,
        MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public AccessibleMemory allocAccessibleMemory(Memory m,
        MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public AccessibleMemory getSubRangeAM(AccessibleMemory m, Address offset,
        Address newLength)
        throws AllocationException, InvalidAddressException;
```

```
    public void freeAccessibleMemory(AccessibleMemory m)
        throws AllocationException;
```

Methods for allocating/freeing DMAMemory

```
    public Memory allocDMAMemory(int mdIdx, MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public Memory allocDMAMemory(MemoryDescriptor m, MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public Memory allocDMAMemory(Address len, MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public Memory allocDMAMemory(AccessibleMemory a, MemoryConstraints c)
        throws AllocationException, InvalidAddressException;

    public void freeDMAMemory(Memory m)
        throws AllocationException;

    public Memory getSubRangeDMA(Memory m, Address offset,
        Address newLength)
        throws AllocationException, InvalidAddressException;
```

Methods for requesting bus manager services

```
/**
     * Ask the bus manager to connect driver interrupt source to the Interrupt
Source Tree.
     */
    public void connectInterruptSource(DeviceInterruptSource devIntSrc);

    /**
    * Ask the bus manager to disconnect our interrupt source from the Interrpt
Source Tree.
     */
    public void disconnectInterruptSource(DeviceInterruptSource devIntSrc);

   /*
    * ask bus for interface source object
    */
    public Object getSerialInterfaceSource();

   /*
    * ask bus for interface source object
    */
    public Object getFrameBufferNativeInterface();
}
```

Interrupts are discussed in a separate section at the end of this chapter.

Annotated DMAMemory Code

```
public Memory allocDMAMemory(int mdIdx, MemoryConstraints c)
       throws AllocationException, InvalidAddressException;
```

With mdIdx, the driver identifies a particular memory descriptor in the array of MemoryDescriptors obtained via getMemoryDescriptors(). Also specified are the MemoryConstraints. The bus manager allocates the specified DMA memory and returns it to the driver.

```
public Memory allocDMAMemory(MemoryDescriptor m, MemoryConstraints c)
    throws AllocationException, InvalidAddressException;
```

A driver cannot actually use this interface, which allows a memory descriptor to be specified directly, rather than indexed to the array. If a driver call this interface, an exception is thrown.

```
public Memory allocDMAMemory(Address len, MemoryConstraints c)
    throws AllocationException, InvalidAddressException;
```

A driver uses the above method to allocate a buffer of DMA memory of specified length and memory constraints.

```
public Memory allocDMAMemory(AccessibleMemory a, MemoryConstraints c)
    throws AllocationException, InvalidAddressException;
```

Drivers use this method to allocate DMA memory, which maps to the same underlying physical memory as the specified accessible memory. The reverse approach is described in the AccessibleMemory allocation section that follows.

```
public void freeDMAMemory(Memory m) throws AllocationException;
```

This method is used to free DMA memory.

```
public Memory getSubRangeDMA(Memory m, Address offset, Address newLength)
    throws AllocationException, InvalidAddressException;
```

Given an already allocated DMA memory, the above method can be used to allocate a sub-range DMA memory object starting at offset, in the original DMA memory object and going for length newLength.

Annotated AccessibleMemory Code

```
public AccessibleMemory allocAccessibleMemory(int mdIdx,
    MemoryConstraints c)
    throws AllocationException, InvalidAddressException;
```

With mdIdx, the driver identifies a particular memory descriptor in the array of MemoryDescriptors obtained via getMemoryDescriptors(). Also specified are the MemoryConstraints. The bus manager allocates the specified accessible memory and returns it to the driver.

```
public AccessibleMemory allocAccessibleMemory(MemoryDescriptor m,
    MemoryConstraints c) throws AllocationException,
    InvalidAddressException;
```

A driver cannot actually use the above interface, which allows a memory descriptor to be specified directly, rather than indexed to the array. If a driver call this interface, an exception is thrown.

```
public AccessibleMemory allocAccessibleMemory(Address len,
    MemoryConstraints c) throws AllocationException,
    InvalidAddressException;
```

The above method can be used to allocate a buffer of regular memory of specified length and memory constraints.

```
public AccessibleMemory allocAccessibleMemory(Memory m,
    MemoryConstraints c) throws AllocationException,
    InvalidAddressException;
```

The above method can be used to allocate AccessibleMemory corresponding to the same underlying memory as DMAMemory (Memory). However, this method is currently not supported. If a driver wants such memory (memory which is double mapped in DMA space and also mapped to be accessible), it needs to allocate the DMAMemory first and then allocate corresponding AccessibleMemory. This is basically the reverse order of the DMA method described above.

```
public AccessibleMemory getSubRangeAM(AccessibleMemory m,
    Address offset, Address newLength) throws
    AllocationException, InvalidAddressException;
```

Given already allocated AccessibleMemory, the above method can be used to allocate a sub-range of the AccessibleMemory object starting at offset in the original AccessibleMemory object and going for newLength.

```
public void freeAccessibleMemory(AccessibleMemory m)
    throws AllocationException;
```

The above method may be used to free Accessiblememory.

Allocating Memory

Following are code fragments for allocating DMAMemory, AccessibleMemory, and memory subranges.

Allocating DMAMemory

```
// Allocate the DMA memory and map to AccessibleMemory

try {
MemoryConstraints dmaMemCons = bus.allocGenericMC(“dma”);

dmaMemory = bus.allocDMAMemory(
    new Address32(largestRx*rxRingSize, 16), dmaMemCons);

MemoryConstraints accMemCons =
    bus.allocGenericMC(PCIBusManager.PCI_MEMNAME);

accMemCons.setCacheMode(MainMemory.CACHE_MODE_INHIBITED);
dmaAccMemory = bus.allocAccessibleMemory(dmaMemory, accMemCons);

} catch (Exception ex) {
System.err.println(
    “Caught Exception while allocating dmaMemory: “ + ex);
ex.printStackTrace();
}
```

Allocating AccessibleMemory.

```
MemoryDescriptor[] mDescs;

try {
constraints = null;

/*
 * Obtain PCI memory descriptors from bus manager.
 */
mDescs = bus.getMemoryDescriptors();

/*
 * Looke for the PCI I/O memory descriptor and allocate accessible
 * memory from it.
 *
 * Can use the devInfo name, or instead use its getSpace() method
 * and check that way.
 */
 int mdIdx;
 for (mdIdx = 0; mdIdx < mDescs.length; mdIdx++) {
    PCIDeviceInfo devInfo =
        (PCIDeviceInfo)mDescs[mdIdx].getDevInfo();
    if (devInfo.getName().equals(PCIBusManager.PCI_IONAME)) {
        constraints = bus.allocGenericMC(PCIBusManager.PCI_IONAME);
            constraints.setAllocAddrs(new Address32(0, 16),
            new Address32(0xFFFF, 16));
        System.out.println(PC3Com: io index is  + mdIdx);
        break;
```

```
    }
}
ioaddr = bus.allocAccessibleMemory(mdIdx, constraints);

} catch (InvalidAddressException e) {
    System.err.println(e);
} catch (AllocationException e) {
    System.err.println(e);
}
```

Obtaining Memory Subranges

```
// Construct packets
Memory subRangeDmaMemory = null;
AccessibleMemory subRangeDmaAccMemory = null;
rxDmaPacket = new PC3ComDmaPacket[rxRingSize];

 for (int i = 0; i < rxRingSize; ++i) {

try {
subRangeDmaMemory = bus.getSubRangeDMA(
    dmaMemory,
    new Address32(largestRx*i) ,
     new Address32(largestRx));

subRangeDmaAccMemory = bus.getSubRangeAM(
    dmaAccMemory,
    new Address32(largestRx*i) ,
new Address32(largestRx));

rxDmaPacket[i] = new PC3ComDmaPacket(
        bus, subRangeDmaMemory, subRangeDmaAccMemory);
} catch (InvalidAddressException e) {
    System.err.println("Caught Exception: " + e);
} catch (AllocationException e) {
    System.err.println("Caught Exception: " + e);
}
}
```

Interrupts

All processes in JavaOS for Business, including the JavaOS Virtual Machine (JVM), run in a single address space. As a result, when the garbage collector is called on to sweep the address space floor, thread dispatching--including interrupt threads--slows measurably. Drivers are constrained from fielding interrupts and processing I/O. To overcome the resulting interrupt latency, JavaOS for Business implements a three-tier interrupt model.

Three-tier Model

The first two interrupt tiers are written in native code, with the first tier representing CPU interrupt-level code. For serial and parallel ports, JavaOS for Business supplies native interrupt handlers, which are linked into the platform. The microkernel oversees the first two tiers of interrupt processing. The third interrupt tier is written in Java and supported by the JavaOSJavaOS for Business Virtual Machine using Java threads.

Interrupt Source Tree

JavaOS for Business represents any device that is capable of interrupting as an *interrupt source*. Because of the possible differences between device and interrupt topologies, a separate class of objects is required to represent the interrupt source.

To pass interrupts to a processor, drivers must supply an interrupt vector number, which is CPU dependent. This vector number is obtained only through low-level access to system resources. To insulate drivers from having to dig too deeply into the operating system--thereby incurring platform dependencies--a bus manager brokers the transaction on behalf of the driver. This is similar to the function performed by bus managers when allocating memory objects for device drivers.

The bus manager again provides a level of abstraction for the driver, this time wrapping interrupt vectors into i*nterrupt source objects* and arranging them as unique objects in a tree structure. Separate interrupt objects are required because separate threads in JavaOS for Business device drivers are used to process interrupts directly from the microkernel. And the microkernel requires interrupt objects in order to track and signal waiting threads. An interrupt tree structure provides the capability to do this in an efficient way.

When a driver starts up, it negotiates with the bus manager for its interrupt source object and then requests that the bus manager plug the object into the interrupt source tree. For each device under the control of a bus manager, at least one entry in the tree is created to represent the interrupt source for that device. The tree is consequently a hierarchy of these interrupt source objects, where the root of the tree represents the CPU itself.

Three Kinds of Interrupt Handlers

JavaOS for Business recognizes *three* levels of interrupt processing, each level defining a handler and an execution context for that kind of handler. Bus interrupt handlers determine the source of the interrupt on a particular bus. Device interrupt handlers process the interrupt.

First Level Handler

The first kind of interrupt handler is a native interrupt handler that executes at CPU interrupt level. All native interrupt handlers are linked into the platform.

A first-level handlers job is to satisfy the immediate real-time needs of a device such as reading data from a device with limited buffering capability. After satisfying the real-time needs of a device, a first-level interrupt handler can cause a second or third-level handler to be queued by the microkernel.

First-level interrupt handlers are the highest priority code in the system, preempting other interrupt handlers and threads. Consequently, the time spent in a first-level interrupt handler should be kept to a minimum.

Second Level Handler

The current implementation of JavaOS for Business does not support second level interrupt handlers.

Third Level Handler

Java interrupt handlers run in the context of Java threads and therefore may use the full resources of the language, the JavaOS for Business framework, and the JDK.

The Java interrupt handler can run in the context of any Java thread. The bus manager either creates one thread per source object *or* allows the driver to create and supply the thread.

A third-level interrupt handler is queued to run if no first or second level handler exists for an interrupt source. The microkernel queues the third-level handler when the device interrupts. A third-level interrupt handler can also run if queued by either a first or second level handler. Third-level interrupt handlers run after first and second-level interrupt handlers, and many other low-priority threads.

A child class of `DeviceInterruptSource` that needs a first level handler should call the native method to fetch the address of the first level interrupt handler. It then calls the `setFirstLevelIntrHandler` method to store this value into the interrupt source tree.

The third level Java based Interrupt handler, `handleThirdLevelInterrupt` has a dummy implementation in the `DeviceInterruptSource` class that is overridden in the child classes. Sample code for this is shown in the following section.

PC3Com Interrupt Processing

The PC3Com driver performs standard interrupt processing that involves the following steps:

- Creating an interrupt source object
- Registering the interrupt object
- Connecting the interrupt source to the interrupt source tree
- Disconnecting the interrupt source

Creating an Interrupt Source Object

An interrupt source object for the PC3Com device is created by extending the base `DeviceInterruptSource` class. Interrupt source methods must be specified and a level handler must be set, as shown in the code below:

`PC3ComInterruptSource` Class

```
package sun.javaos.PC3Com;
import sun.javaos.net.*;
import javax.system.database.Entry;
import javax.system.interrupts.*;

public class PC3ComInterruptSource extends DeviceInterruptSource {
/**
 * Constructor for a leaf device interrupt source.
 */
public PC3ComInterruptSource(String name, PC3Com e) {
    super(name);
    etherlink = e;
    setInterruptLevel(e.getIrq());
}

/**
 * Third-level Interrupt Handler for the device.
 */
public boolean handleThirdLevelInterrupt(long when) {
    etherlink.handleInterrupt();
    return true;
}

private PC3Com etherlink;
}
```

Registering an Interrupt Object

Once the source code for an interrupt class is written, a source object must be instantiated by the driver and registered with the bus manager. To do this, the PC3Com driver requests bus manager services, shown in this fragment of `PC3Com` code:

```
try {
PC3ComInterruptSource pC3ComIntSrc =
    new PC3ComInterruptSource(“PC3Com-0”, this);
bus.connectInterruptSource(pC3ComIntSrc);
} catch (Exception e) {
System.err.println(“PC3Com constructor:  “
    + “initialization failed: “ + e.getMessage());
throw e;
}
```

Connecting to the Interrupt Tree

The next step in the process is linking the object into the interrupt source tree. The `connectInterruptSource` method of the bus manager does this in the following code fragment from the PC3Com driver:.

```
PC3ComInterruptSource pC3ComIntSrc = new PC3ComInterruptSource(
    PC3Com-0, this);

bus.connectInterruptSource(pC3ComIntSrc);
```

The bus manager also creates and starts the interrupt thread.

In the near release, JavaOS for Business will support additional flavors (parameter values) of `connectInterruptSource` so that drivers will be allowed to create threads (or use pre-existing threads) on their own, without bus manager intervention. Drivers may also place the desired or required interrupt level in the source object. Currently, The interrupt level is set in the source by the bus manager, as shown in the method below.

Now that the `PC3ComInterruptSource` is constructed and passed to the `connectInterruptSource` method of the bus manager, the PC3Com driver is set up to receive and process interrupts.

`getIntLine` *Method*

This PC3Com method requests bus manager services to determine the interrupt vector of the device. The irq value is placed in the source by the bus manager.

```
    irq = ((PCIBusManager)bus).getIntLine();
```

Disconnecting from the Interrupt Tree

The `disconnectInterruptSource` method undoes what the `connectInterruptSource` method did. In other words, the method: (1) stops the thread; and (2) removes the interrupt source from the tree.

`DeviceInterruptSource` Class

This is the base interrupt class. It encapsulates all:

- management of interrupts for a device
- processing of interrupts for a device

PC3Com extends this class and links in a third-level interrupt handler.

`DeviceInterruptSource` Methods

The `DeviceInterruptSource` class supports the following interface methods:

- `setFirstLevelIntrHandler(int firstLevelIntrHandler)`
- `setSecondLevelIntrHandler(int secondLevelIntrHandler)`
- `handleThirdLevelInterrupt(long when)`
- `getInterruptLevel()`
- `setInterruptLevel(int level)`
- `getInterruptThread()`
- `setInterruptThread(Thread interruptThread)`
- `setFirstLevelIntrHandler()`
- `setDflHandler (int handler)`

Defining a Device-specific Interrupt Source

To define a specific interrupt source, a driver typically extends `DeviceInterruptSource.` All the PC3Com driver does to the base `DeviceInterruptSource` class is link in its 3rd-level interrupt handler, as shown below:

```
(from PC3ComInterruptSource)

public class PC3ComInterruptSource extends DeviceInterruptSource {
    /**
     * Constructor for a leaf device interrupt source.
     */
    public PC3ComInterruptSource(String name, PC3Com e) {
        super(name);
```

```
        etherlink = e;
        setInterruptLevel(e.getIrq());
    }

    /**
     * Third-level Interrupt Handler for the device.
     */
    public boolean handleThirdLevelInterrupt(long when) {
        etherlink.handleInterrupt();
        return true;
    }

    private PC3Com etherlink;
}
```

`handleInterrupt` Override Method

The `DeviceInterruptSource` class declares a method without specifying a device for the method to act.

```
public boolean handleThirdLevelInterrupt(long when) {
return true;
    }
```

PC3Com overrides this method to specify a third-level interrupt handler, as shown below. When an interrupt occurs, the `handleInterrupt`() method of the PC3Com driver is invoked.

```
public boolean handleThirdLevelInterrupt(long when) {
        etherlink.handleInterrupt();
        return true;
```

`etherlink` is the PC3Com driver.

`DeviceInterruptSource` *Code*

Following is the complete source code for the `DeviceInterruptSource` class:

```
package javax.system.interrupts;

import javax.system.database.Entry;

public class DeviceInterruptSource extends InterruptSource
 implements  DeviceInterruptManager, Runnable {

    protected int firstLevelIntrHandler; //Stores native pointer to the first
                                         //level handler if any for this
                                         //interrupt source
   protected int secondLevelIntrHandler; //Stores native pointer to the second
```

```
                                        //level handler if any for this
                                        //interrupt source
protected int interruptLevel; //The IPL at which this source interrupts
protected Thread interruptThread; //Reference to Interrupt Handler Thread
                                  //dedicated to this Interrupt Source.
/**
 * Constructor for a leaf device interrupt source.
 */
public DeviceInterruptSource(String name) {
    super(name, 0);
    interruptLevel = DeviceInterruptSource.DontCareWhichLevel;
}

/**
 * Constructor for a parent (bus) device interrupt source.
 */
public DeviceInterruptSource(String name, int maxChildSources) {
    super(name,maxChildSources);
}

//Now implement the DeviceInterruptManager interfaces

public boolean setFirstLevelIntrHandler(int firstLevelIntrHandler) {
    this.firstLevelIntrHandler = firstLevelIntrHandler;
    return true;
}

public boolean setSecondLevelIntrHandler(int secondLevelIntrHandler) {
    this.secondLevelIntrHandler = secondLevelIntrHandler;
    return true;
}

/**
 * Third-level Interrupt Handler for the device.
 * Derived InterruptSource classes for drivers will override
 * this method.
 */
public boolean handleThirdLevelInterrupt(long when) {
   return true;
}

public int getInterruptLevel() {
    return interruptLevel;
}

public void setInterruptLevel(int level) {
    interruptLevel = level;
}

public Thread getInterruptThread() {
    return interruptThread;
}

public void setInterruptThread(Thread interruptThread) {
```

```
        this.interruptThread = interruptThread;
    }

/** the time stamp is read wehn the hardware interrupts. The time is calculated
in microseconds and passed up as a parameter to the third level interrupt*/
public void run() {
        while (true) {
            try {
                    long timeStamp = waitForNextInterrupt();
                handleThirdLevelInterrupt(timeStamp);
            } catch (Throwable e) {
                try {
                    e.printStackTrace();
                } catch (OutOfMemoryError m) {
                }
            }
        }
    }

        /**
         * Start this device. Called by the device driver this will
         * typically involve enabling the interrupts for this source
         * and starting the third level interrupt handling thread
         * that was created by the Bus Manager for us.
         *
         * This method is not currently used.
         */
    public void start() {
    //enable the interrupts for this level.
        interruptThread.start();
    }

        /**
         * Stop this device. Called by the device driver this will
         * typically involve disabling the interrupts for this source
         * and stopping the third level interrupt handling thread
         * thstampat was created by the Bus Manager for us.
         *
         * This method is not currently used.
         */
    public void stop() {
        //disable the interrupts for this level.
        interruptThread.stop();
    }

       //XXX This method should go away. But needs to be there until the Uart
        //driver starts using setFirstLevelIntrHandler() method.
    public void setDflHandler (int handler) {
        firstLevelIntrHandler = handler;
    }

    //Implement InterruptManagement interface that is extended by
        //the DeviceInterruptManager interface we implement
    public native boolean enableInterrupt();
```

```
    public native boolean disableInterrupt();
    public native void acknowledgeInterrupt();

    public native long waitForNextInterrupt();

    public static final int DontCareWhichLevel = -1;
}
```

Interrupt Thread Creation

In the current JavaOS for Business version, interrupt threads are created by bus managers in the following manner:

1. The driver `PC3ComInterruptSource` class extends `DeviceInterruptSource` and uses a constructor to create an interrupt source.
2. The driver then calls `bus.connectInterruptSource` to register the interrupt source with the bus manager.
3. The bus manager places the source object in the interrupt source tree.
4. The bus manager then creates a thread associated with the interrupt source object on behalf of the driver.
5. The bus manager starts the thread.
6. This thread waits for the third level interrupt signal to be issued by the microkernel as a result of some device activity (an event).
7. The thread wakes up and interrupts.

In the next release of JavaOS for Business, the architecture will expand to include more flexibility for drivers in the creation of interrupt threads. In the procedure above, for example, the driver will have the option in steps 4 though 7 of creating the thread itself, rather than having to rely on the bus manager to create the thread for it. The bus manager will communicate with the driver after the source object is placed in the tree, and the driver can then take over thread creation on its own.

A

Appendix A: JDDG Addition

JavaOS Boot Interface (JBI)

This section contains technical details on the JavaOS Boot Interface (JBI), and is intended specifically for licensees who plan to customize the booter to suit a particular need or business environment.

The information contained in this section serves to supplement, not replace, the information covered in the *JavaOS for Business Reference Manual*, Chapter 6, JavaOS Boot Interface. It is recommended the reader be familiar with the material found in Chapter 6, before proceeding on with this section.

While the text below makes mention of some JavaOS for Business classes and methods, all programs, subroutines, and functions provided in this section are in C language.

Booter

The JBI is the C language native interface and code execution environment required to boot the JavaOS for Business operating system. The booter is the program or series of programs designed to initialize the JavaOS for Business platform and to bootstrap the operating system.

The *Microkernel* is the *target* software the booter loads and starts. Like other kernels, the Microkernel needs to know what computing environment exists when booted, and how to take over the management of the environment from the booter.

JavaOS for Business is targeted at a wide range of products, from full featured legacy computers, to Network Computers (NCs). The JBI gives product designers and implementation teams a great deal of flexibility when booting JavaOS for Business.

Some of the typical booting choices include:

- Network booting of an NC from a server
- Flash RAM/ROM or Disk booting of an NC
- ROM booting of an embedded device

JBI

All boot operations are normalized to a standard interface called the JavaOS Boot Interface (JBI). This interface is used both by the Microkernel and the JDK Runtime layers of JavaOS for Business.

The JBI defines a bi-directional interface as well as an OS execution environment. The bi-directional interface is composed of the following:

- a function that passes control to JavaOS for Business
- an interface that allows JavaOS for Business to request services of the booter system

These two sub-interfaces are known as the *Start-up* and *Boot Operations* Interfaces.

The following illustration depicts the relationship between JavaOS for Business and these two interfaces:

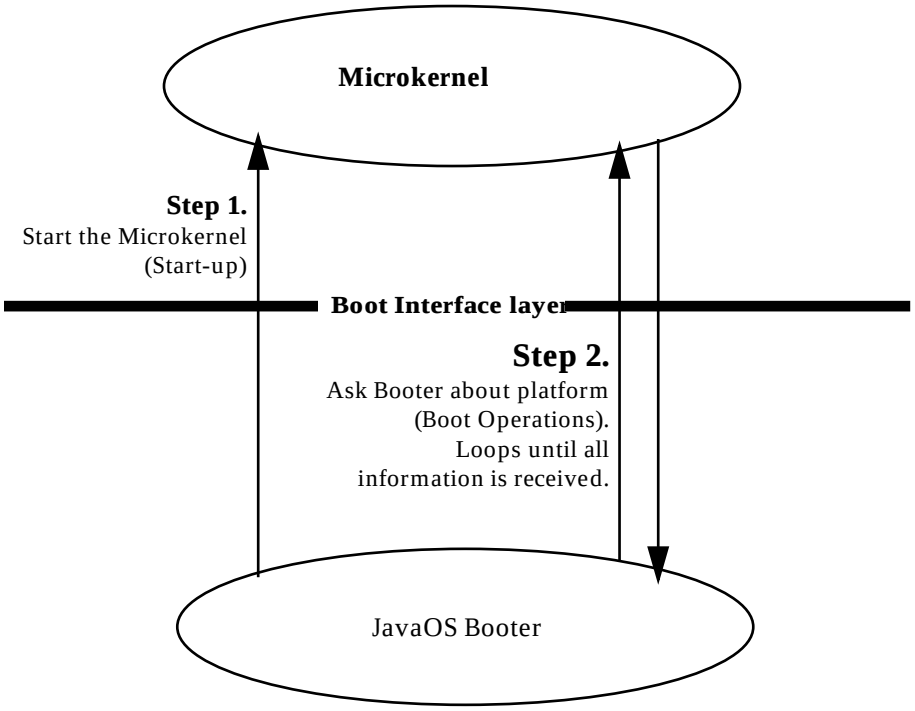

FIGURE A-1 JBI Booter and Microkernel Interface

The JBI is designed to give product designers flexibility when making booting decisions. Some JavaOS for Business booter implementations may leverage an existing PROM standard such as OpenBoot (IEEE 1275-1994) or a PC-BIOS.

Some embedded products, such as a smart phone, may require a complete custom PROM. Regardless of the PROM composition, all customization and implementation is abstracted from the Microkernel using the JBI.

The following figure illustrates some possible booting systems:

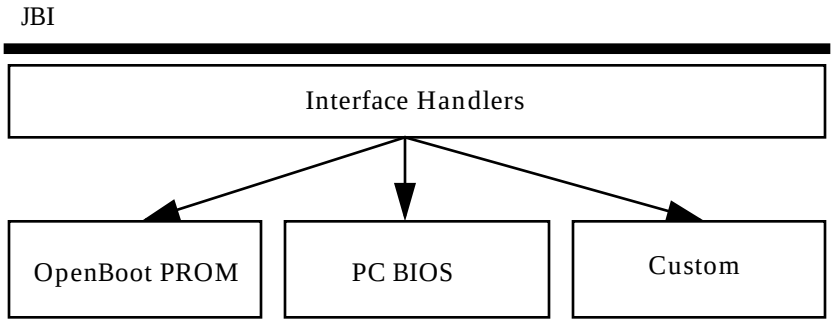

FIGURE A-2 Booting Systems

Start-up Interface

The JavaOS for Business *Start-up Interface* consists of a single function that transfers control from the booting system to JavaOS for Business. The following information is passed as parameters from the booting system to JavaOS for Business:

- **BootContextID**. An ID that identifies an execution context for the booting system. This ID, its value, and the represented execution context are 100% owned by the booting system. JavaOS for Business stores this ID until required by the booting system. Booting systems that maintain their own execution context may simply pass a null. Other booters may wish to pass a pointer to the booter's global data.
- **BootOps Pointer.** A pointer to a structure (`JBIBootOps`) whose members are suitable for invocation by JavaOS for Business. Each `JBIBootOps` function requires that the `BootContextID` be passed as the first parameter. Each function follows the C calling conventions of the CPU.
- **Initial Stack Address.** A pointer to the first byte of the memory region comprising the *initial stack* on which the Microkernel is placed before the booter transfers control to the Microkernel. The Microkernel uses this stack for its initial bootstrap functions.
- **Initial Stack Size.** The size, in bytes, of the initial stack.

Note: When a Microkernel returns to the booter (i.e., caller of the start-up entry point) an *error condition* occurs. However, booters must report this event should it occur and enter some recoverable state.

BootContextID

The Microkernel makes a series of function call-backs to the booter. The first parameter of each call is the `BootContextID`. The typical boot system uses the `BootContextID` to locate the boot system's global booter data. The following figure illustrates this concept.

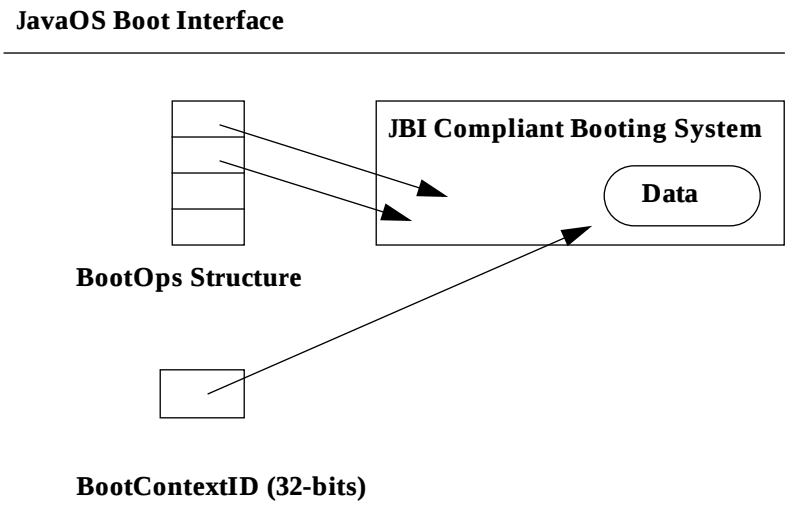

FIGURE A-3 BootContextID Usage

OS Execution Environment

The execution environment required by JavaOS for Business is the C language JDK Runtime environment defined for the target CPU. That is, the Microkernel expects to execute as a C program would on the CPU.

The JavaOS for Business binary image defines the `startJavaOS` text symbol. The booter calls this symbol as a C language subroutine passing the `contextID` and a pointer to the `BootOps` structure as parameters.

The following C language types and function prototypes are used to define the start-up entry point to the Microkernel. The `BootContextID` is defined as an unsigned 32-bit value:

```
typedef unsigned long JBICid;
```

The boot operations structure is defined below. A pointer to this structure is supplied as the second argument to `startJavaOS for Business`.

```
typedef struct {

    long (*jbiGetVersion)(JBICid cid);
    long (*jbiQuiesce)(JBICid cid);
    long (*jbiDebugPrintf)(JBICid cid, const char *format,
    ...);

    long (*jbiGetFile)(JBICid cid, char *fileName, char
    *buffer, int size);
    jbiExtFunctionTable *(*jbiGetExtFunctions)(JBICid cid);

    JBIFileList (*jbiGetFileList)(JBICid cid);
    JBIStatusCode (*jbiGetBootStatus)(JBICid cid, JBIBootOpID
    opid, void **data);
    unsigned char *(*jbiGetPlatformID)(JBICid cid);
    unsigned char *(*jbiGetMachineD)(JBICid cid);
    void *(*jbiGetBootData)(JBICid cid);
    JBIReboot (*jbiGetRebootVector)(JBICid cid,unsigned int
    *extended);

    long (*jbiGetPhysicalMemoryMap)(JBICid cid, JBIPhysMemoryMap
    **map);
    long (*jbiGetVirtualMemoryMap)(JBICid cid, JBIVirtMemoryMap
    **map);

    JBIDevTreeEntry (*jbiDevTreeGetRootEntry)(JBICid cid);
    JBIDevTreeEntry (*jbiDevTreeGetBootEntry)(JBICid cid);
    JBIDevTreeEntry (*jbiDevTreeGetDebugEntry)(JBICid cid);
    JBIDevTreeEntry (*jbiDevTreeGetParentEntry)(JBICid cid,
    JBIDevTreeEntry entry);
    JBIDevTreeEntry (*jbiDevTreeGetFirstChildEntry)(JBICid cid,
    JBIDevTreeEntry entry);
    JBIDevTreeEntry (*jbiDevTreeGetPeerEntry)(JBICid cid,
    JBIDevTreeEntry entry);
```

```
long (*jbiDevTreeGetNameLength)(JBICid cid);

long (*jbiDevTreeGetNextProperty)(JBICid cid, JBIDevTreeEntry
entry, const char *prevPropName, char *nextPropName);

long (*jbiDevTreeGetValueLength)(JBICid cid, JBIDevTreeEntry
entry, const char *propName);

long (*jbiDevTreeGetPropertyValue)(JBICid cid,
JBIDevTreeEntry entry, const char *propName, void *value,
long length);

long (*jbiDevTreeSetPropertyValue)(JBICid cid,
JBIDevTreeEntry entry, const char *propName, const void
*value, long length);

} JBIBootOps;
```

The JavaOS for Business entry point is then prototyped as:

```
void startJavaOS(JBICid id, JBIBootOps *jbis, void *stackAddr,
unsigned long stackLength);
```

Boot Operations Interface Overview

The boot operations interface (*BootOps*) is a set of C language subroutines invoked by the Microkernel during the Microkernel's booting phase. All booting systems must implement each BootOps function.

At a minimum, for a given boot operation, a valid function pointer must be provided even if it does nothing more than return an error code. Null pointer values are not allowed.

BootOps functions are invoked by the Microkernel using its current stack. The booter must ensure that the amount of stack space used (for other function calls, local variables, etc.) is kept to a minimum to avoid stack overflow, a fatal error.

Each release of the JBI is assigned a version number. Each subsequent version builds upon the collective functionality of all existing functions. At compile time, the constant `JBI_VERSION` in the `jbi.h` header file defines the current version number. Booters return this value, and Microkernel implementations use it to compare the known version with that returned by the booter.

The clients of the `BootOps` interface are all composed of native code; primarily the Microkernel, but also other modules as warranted by the product composition. The `BootOps` interface is published to the Microkernel via a pointer to a structure of C subroutine addresses.

The Microkernel contains a set of *cover functions* that perform the subroutine invocation on behalf of clients. The set of cover functions is collectively called the *BootOps C Library.* The BootOps C library *binds* the Microkernel and all other native clients to the functions published by the booter.

This architecture serves two purposes.

- First, all clients are abstracted from the `BootOps` structure address, its location, size, and the `BootContextID`.
- Secondly, client invocation of boot operations can be coded as simple subroutine calls.

The following figure illustrates the call path from a native client to the boot operations implementation.

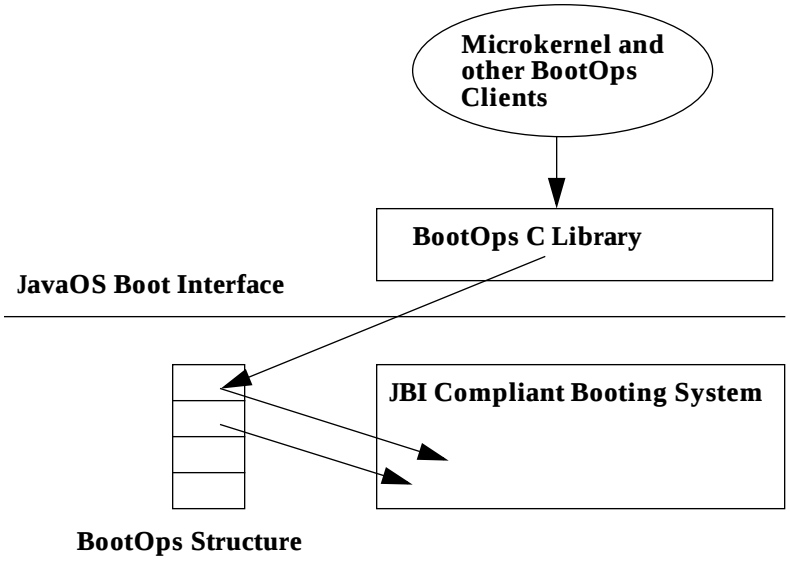

FIGURE A-4 BootOps Invocation Flow of Control

The `BootOps` interface is designed to be transitory in nature. That is, the complete interface *implementation* can be transferred from the booter to the Microkernel. This is useful when the Microkernel has finished using the boot operations, but when other clients still require the functionality. The switch is accomplished by replacing the published functions and shutting down the booter's implementation with a special `BootOps` function (`jbiQuiesce`) invocation.

BootOps Interface Service Categories

The `BootOps` interface is a set of C language subroutines providing the following categories of services:

- Platform physical memory information.
- Platform virtual memory information.
- Platform and Machine IDs.
- Booter's Device discovery information.
- Access to files down-loaded during the booting process.
- Booter status.
- Reboot vector access.
- Booter specific data access.

When the Microkernel invokes a `BootOps` function, it uses its own stack. The `BootContextID` is provided to the booter so that it addresses the information relevant to this particular instance of the callback.

Physical Memory

The Microkernel calls the booter to obtain the map of physical memory.

The physical memory map consists of a series of entries, each entry describing a range of physical memory. The number of map entries is variable in length, but the entire potential physical memory address space must be accounted for in the physical memory map, including any invalid regions (i.e., memory holes).

Each entry *types* the range of memory. The type conveys the memory's intended or current use such as ROM, RAM, not valid, and so on.

The following C enumerated type defines the possible range types.

```
typedef enum {
rangeIsNotValid,/* Range not usable (i.e. hole) */
rangeIsROM,/* Read-only or flash memory */
rangeIsRAM,/* Physical RAM */
rangeIsFixedIO,/* Memory for permanent I/O devices */
rangeIsExpansonIO,/* Memory for dynamic I/O devices */
rangeIsPlatformControl,/* Platform specific control */
rangeIsFileData,/* Downloaded File data */
rangeIsCustom/* Custom range name */
} JBIPhysMemoryRangeType;
```

JavaOS for Business never attempts to access physical memory ranges defined as `rangeIsNotValid`.

A generic content type is also specified for each range. The content is defined as either free memory, booter memory, booter-to-Microkernel data passing memory, or Microkernel memory.

```
typedef enum {
rangeIsFree,/* Free for JOSB Microkernel use */
rangeIsBooter,/* In use by booter until acquiesced */
rangeIsKernel,/* Portion of JOSB loaded here */
rangeIsData,/* Data from booter to Microkernel is here */
} JBIPhysMemoryRangeContent;
```

Finally, the actual memory associated with the range is specified using a starting address and a length in bytes.

```
typedef struct {
void *paddr;/* Starting physical address*/
unsigned long length;/* Length of range in bytes */
} JBIPhysMemoryRange;
```

A physical memory map entry, fully describing a single range of physical memory, puts all of these enumerated types together.

```
typedef struct {
JBIPhysMemoryRangeType type;
JBIPhysMemoryRangeContent content;
JBIPhysMemoryRange range;
} JBIPhysMemoryMapEntry;
```

A complete physical memory map is comprised of a contiguous array of memory map entries, and a count of the number of entries.

```
typedef struct {
unsigned long numEntries;/* Number of entries in map */
JBIPhysMemoryMapEntry entry[1];/* Array of map entries */
} JBIPhysMemoryMap;
```

The booter is responsible for allocating the memory for the map within its own range of memory (`rangeIsBooter`). A map pointer is returned to the Microkernel. The ranges in the map must appear in ascending order based on the starting physical memory address of the range.

As an example, consider the hypothetical system depicted in the *Physical Memory Map Example* figure.

The booter itself is occupying a portion of memory as well as the Microkernel image. There is a small memory hole at 0x9f000. Only 64MB of RAM is installed, so there is a large memory hole starting at 0x4000000 which must be represented by an entry in the physical map.

Additionally, a PROM is accessible via the upper 2MB of the physical address space, and just below it lies 14MB of memory mapped I/O addresses.

There are ten map entries denoting the following features associated of the range of physical memory:

- range type
- current usage of the range (free, booter, Microkernel)
- starting address and length

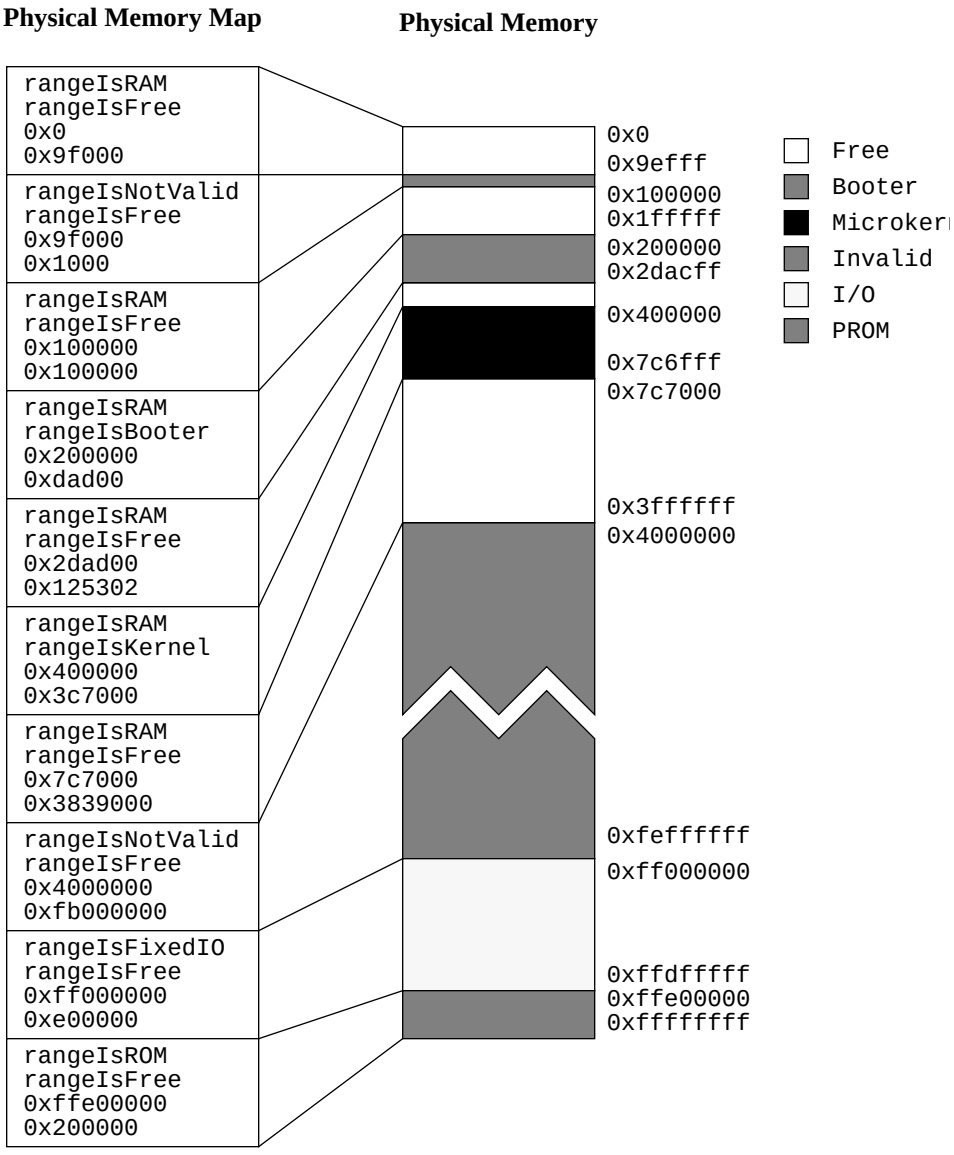

FIGURE A-5 Physical Memory Map Example

Virtual Memory

The Microkernel also calls the booter to obtain the map of virtual memory.

The virtual memory map communicates the current MMU configuration to the Microkernel. As with the physical memory map, a number of C data structures and enumerated types have been defined.

It is common for an MMU to have the ability to map pages with individual caching modes (uncached, write-through, write-back). This attribute is specified for a virtual memory map entry using the following enumerated type:

```
typedef enum {
rangeIsUncached,
rangeIsCachedWriteThrough,
rangeIsCachedWriteBack
} JBIVirtMemoryRangeCaching;
```

The address range represented by the virtual map entry is defined with the following structure:

```
typedef struct {
void *vaddr;/* Starting virtual address */
void *paddr;/* Starting physical address */
unsigned long length;/* Length of range in bytes */
} JBIVirtMemoryRange;
```

A virtual memory map entry describes a range that is both virtually and physically contiguous. The cache mode applies to the entire range.

For a virtually contiguous area of memory that is not physically contiguous, multiple virtual memory map entries are required to describe each physically contiguous portion. A worse case scenario would be where each MMU page is represented by its own entry, though this should never be the case in reality.

A complete virtual memory map is a contiguous array of virtual memory map entries, and a count of the number of entries.

```
typedef struct {
unsigned long numEntries;/* Number of entries in map */
JBIVirtMemoryMapEntry entry[1];/* Array of map entries */
} JBIVirtMemoryMap;
```

The booter is responsible for allocating the memory for the map within its own range of memory (`rangeIsBooter`). A map pointer is returned to the Microkernel. The ranges in the map must appear in ascending order based on the starting virtual memory address of the range.

Unlike the physical memory map, which must account for all potentially addressable physical memory, the virtual memory map describes only the virtual-to-physical mappings that have been setup in the MMU by the booter.

If no MMU hardware is present, then the virtual memory map consists of entries that identity-map (virtual = physical) the booter and Microkernel physical memory ranges. In this case, the map is still used only to indicate possible range-specific caching modes.

Considering the memory configuration described in the *Physical Memory Map Example* figure, assume the booter has identity-mapped itself, the I/O address range, and the PROM, and that the Microkernel has been mapped according to the virtual addresses specified in the executable file header.

If the Microkernel text is mapped in at 0xf8000000 and its data and BSS (uninitialized data area), are mapped to 0xf8600000, then, assuming the write-back caching attribute holds, the virtual memory map would look like:

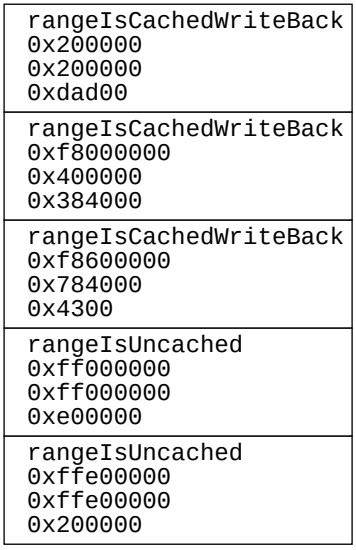

FIGURE A-6 Virtual Memory Map

Booter Memory

Once the booter has invoked the `startJOSB` Microkernel entry point, the booter may no longer change its use of physical or virtual memory resources.

Specifically, in the case of the booter dynamic memory usage, the booter can be managing dynamic memory within one or more `rangeIsBooter memory` ranges in response to JavaOS for Business invoked boot operations, but the locations and sizes of the ranges cannot be changed.

Before invoking the Microkernel, the booter must make these ranges of a sufficient size to handle all boot operations. If the booter itself makes callbacks into other boot components (BIOS, Open Firmware, etc.), it must ensure it has sufficient memory allocated ahead of time for these calls.This is necessary in order to avoid memory usage conflicts between JavaOS for Business and the booter.

After obtaining the memory maps from the booter, JavaOS for Business may begin to make use of free memory regions, even *before* it acquiesces the booter.

Initial JavaOS for Business Microkernel Environment

In addition to the Microkernel `startJavaOS` entry point and the explicit JBI API entry points, a number of other important expectations of both the booter and the Microkernel are made.

Initial Microkernel Stack

The booter must give the Microkernel an initial stack, the top of which is provided in the stack pointer register appropriate for the C-calling convention of the native CPU's Instruction Set Architecture (ISA).

The starting address and length of the memory for the stack is provided to the Microkernel via the third and fourth parameters to `startJOSB`. These parameters are defined as:

`stackAddr`	The first byte (lowest address) of the memory range containing the stack.
`stackLength`	The size of the stack in bytes.

The last usable byte of the initial stack range is at the address:

`stackAddr + stackLength - 1`.

It is recommended that the initial stack be a multiple of the CPU page size, but at a minimum it must be 32-byte aligned.

A stack size of a minimum of 16KB is expected to provide ample space for invoking Microkernel functions, as well as call-backs to the booter to obtain the memory maps.

The following figure indicates the relationships of the stack address, length, and top of stack as defined by the ISA. Configuration (A) represents an architecture where the stack pointer is decremented to push new elements on to the stack, and (B) indicates an architecture where the stack pointer is incremented for a push operation.

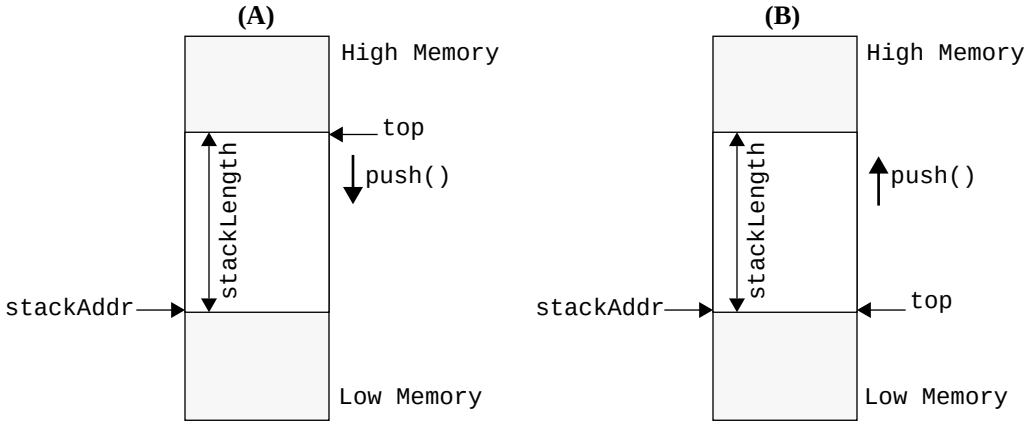

FIGURE A-7 Stack Address Relationships

For the above two configurations, the following relationships hold:

```
(A)          top = stackAddr + stackLength - 1

(B)          top = stackAddr
```

The booter may place information on top of the initial stack (e.g., parameters to `startJavaOS`) pertinent to the Microkernel. However, the Microkernel may modify any portion of the initial stack memory.

Therefore, the booter is not allowed to place non-volatile information, such as the `BootOps` structure itself, within the initial stack. The initial stack memory has the content definition of `rangeIsKernel` within the physical memory map.

The initial stack is provided primarily for the task of early Microkernel bootstrap and calling back to the booter to obtain the physical and virtual memory maps. After this callback, the Microkernel establishes its own memory management mechanisms and moves to another Microkernel stack. At this time, the memory associated with the initial stack may be used by the Microkernel for other purposes.

Device Interrupt Status

The booter must ensure that all device interrupts are disabled prior to invoking `startJOSB`. Additionally, all I/O (DMA, Programmed I/O, etc.) must be suspended before the Microkernel is entered.

Microkernel Initialization

The Microkernel must zero its own BSS (uninitialized data area), at the proper point in its bootstrap code. The booter is not required to perform this task on behalf of the Microkernel.

Only after the booter has been acquiesced (`jbiQuiesce`), can the Microkernel reclaim any memory in the physical memory map of the content `rangeIsBooter`. Callbacks to the booter after this point may result in aberrant behavior.

File Access Information

A JBI-compliant booter may provide the Microkernel optional access to a set of files downloaded by the booter itself. When implemented, the booter and boot server work together to download a variable set of files that may augment, enhance, or describe the required binary JavaOS for Business image.

This feature is primarily designed to download Java classes, packages, and services such as device drivers. When the Microkernel begins to initialize the JavaOS for Business JDK Runtime, these downloaded files are automatically appended to the JavaOS for Business ROM file system for easy access by the JVM class loader.

Memory used by the booter to store downloaded files should be marked as `rangeIsFileData` (specific data type) and `rangeIsData` (generic data type).

The memory allocated by the booter to hold the `JBIFileList` is released upon the `JBIQuiesce` call. The memory to hold the file data however, (`rangeIsFileData`), is not released and therefore accessible by the Microkernel during the initialization of the ROM file system.

The set of downloaded files is described using variable length data structures called the `JBIFileList`. The structures contain a count of downloaded files and an array of `JBIFileDesc` structures that describes a file.

The `JBIFileList` structure is defined as follows:

```
typedef struct {
unsigned long numFiles; /* number of file descriptors */.
JBIFileDesc files[1]; /* array of file descriptors */
} JBIFileDesc;
```

The `JBIFileDesc` structure is defined as follows:

```
typedef struct {
unsigned char *name; /* the file name */
unsigned char *data; /* the file's contents */
unsigned long length; /* the length of the file's contents
*/
} JBIFileDesc;
```

- where *name* = an ASCII null-terminated string containing a file URL. When retrieving file status information, this field may be null. A null value indicates that an error occurred during the download.
- where *data* = a pointer to the file's data. Null indicates a download error.
- where *length* = the number of data bytes.

A file can also be downloaded by the `jbiGetFile`. In this case, a name of the file and memory to put in is passed to the booter and the booter puts that file from the server in the memory specified.

Support for Extended Functions

In order to support additional functions which might be required for certain JavaOS for Business implementations, the `jbiGetExtFunctions` return a pointer to the `JBIExtFunctionTable`. The JBI Extended Functions and JBI Extended Function Table define the number of extended functions supported by a client.

The `JBIExtFunction` structure is defined as follows:

```
typedef struct{
  int FunctionID;
  long (*jbiExtFunction)();
}jbiExtFunction;
```

- where `FunctionID` = A unique integer value identifying the `jbiExtFunction`
- where `jbiExtFunction` = The function itself.

The `jbiExtFunctionTable` structure is defined as follows:

```
typedef struct{
  unsigned long numEntries;
  jbiExtFunction entry[];
}jbiExtFunctionTable;
```

- where `numEntries` = Number of Extended Functions which is a count of `jbiExtFunction` structures that follow
- where `entry[1]` = The first `JBIExtFunction`

Booter Operation Status

JBI booters support an expanded error and status reporting mechanism that augments the negative (<0) return value errors.

The additional status available pertains to machine status and error recovery. Errors that would cause the boot system to fail are not reported through this mechanism.

The `jbiGetBootStatus` call returns a status code and data given a booting operation ID as input.

The `JBIStatusCode` codes are as follows:

TABLE 1-1 JBI Status Codes

Status Code	Meaning
statusSuccess	The boot operation identified by the opID was successful.
statusInfo	The boot operation has further information available. The data returned contains the information.
statusWarning	The boot operation has warning information available. The data returned contains the information.
statusFail	The boot operation failed. The data returned describes the failure.
statusUnsupported	The attempted boot operation is not supported by this implementation of the booter.
statusInvalid	JBI operation is not valid.
statusCustom	Implementation - dependent error or status condition and associated data

Platform Device Information

Platform device information is conveyed from the booter to the Microkernel using an abstract data structure called a *device tree*. The device tree is a data structure that the JavaOS for Business boot system creates to provide information about physical devices available to JavaOS for Business.

The following figure depicts the hierarchical nature of the device tree.

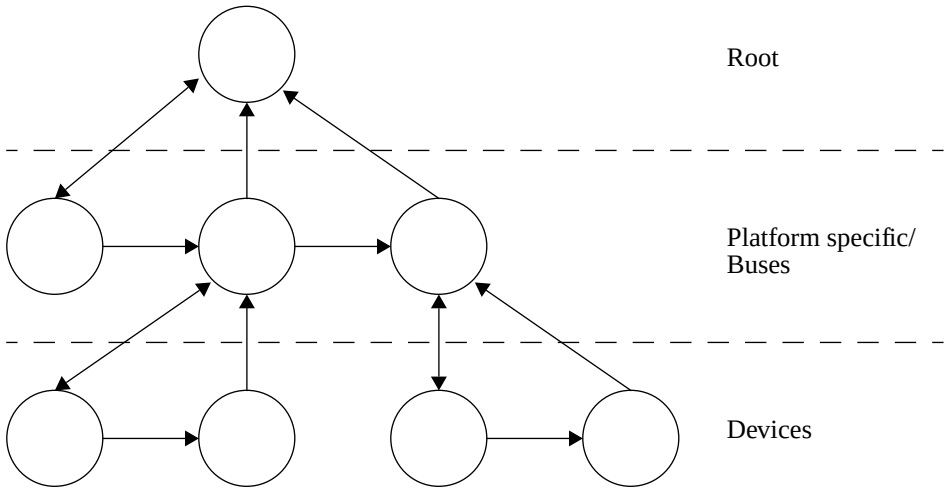

FIGURE A-8 Device Tree Hierarchical Data Structure

The top-level entry is the *root* of the tree. The direct children of the root represent system level buses (PCI, SBus, ISA, etc.) or platform specific configuration entries (e.g., options).

The third level consists of physical devices (SVGA, network adaptor, SCSI host adaptor, etc.).

Each individual entry in the device tree contains references to its neighbor entries, thus establishing the tree hierarchy. In addition, each entry refers to one or more *properties*, or name-value pairs of entry specific information.

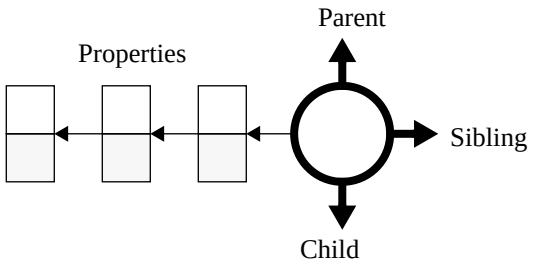

FIGURE A-9 Device Tree References

There are two property connections:

- property name
- property value

The *property name* is an ASCII string. The allowable characters are those defined by the IEEE 1275-1994 standard:

- Name consists of one to thirty-one printable characters.
- Names shall not contain uppercase characters, or the characters
  -)
  - \
  - :
  - [,]
  - @
- Property names beginning with the character + are reserved for use by future revisions of the IEEE 1275 standard.

The *property value* is an array of zero or more bytes. A zero length property value is typically used when a given property represents a boolean state with its presence (true) or absence (false). Otherwise, the value can represent a sequence of bytes, a null-terminated text string, a 32-bit integer, or a composite of all of these types.

Values are represented in big-endian format.

At a minimum, each entry in the device tree must have a name property, where the value is a null terminated text string.

The JavaOS for Business Platform Manager and Bus Manager make use of the properties of various entries in the device tree. For additional information on the JavaOS Driver Interface (JDI), please refer to the *JavaOS for Business Technical Reference Manual*.

However, the platform and bus managers can be specific to a JavaOS for Business implementation. See Standard Device Properties section in this section for more information.

Some PROMs such as OpenBoot can probe for devices and build the tree in a very dynamic fashion. JBI booters layered upon OpenBoot rely on the OpenBoot PROM to do all the work of tree building.

Other booters may take a simpler approach and just hard-code the tree, or maintain it in flash memory, or other local media. It is also possible for the booter to provide little in the way of device information, and then for the JavaOS for Business platform and bus managers to perform their own device discovery.

Device discovery is highly platform-dependent and is isolated from the upper layers of JavaOS for Business by the booting system.

As subsequent layers of JavaOS for Business start, the JBI is used to examine and ultimately to copy the tree from the booting system's memory into the Java JDK Runtime heap as a collection of device entry objects. The JavaOS for Business I/O system may edit or append information to the device tree during this copy phase.

Once the tree is copied into the Java heap, the I/O system begins finding bus managers and device drivers for each entry in the tree. This process is called *driver matching*.

When the device tree is fully copied into the JavaOS for Business JDK Runtime layer, a *quiesce* call is made to the booter to indicate the completion of the booting phase of JavaOS for Business.

The following C typedef is used to identify an entry in the device tree.

```
typedef void * JBIDevTreeEntry;
```

Device entry IDs should not be interpreted. Each booting system is allowed to implement their own ID mechanism. The most common ID mechanism is a pointer or a handle.

Boot Operations Function Calls

The JBI `BootOps` function calls fall into the following categories:

- Miscellaneous functions
- File access functions
- Memory management functions
- Device tree navigation functions
- Device tree property functions
- Extended functions

The `BootOps` interface is documented for clients in terms of the C library cover functions. These functions are used to abstract from clients the `BootOps` structure pointer and the `BootContextID`.

Both the `BootOps` structure pointer and the `BootContextID` are parameters provided by the booter when invoking the `startJavaOS` Microkernel entry point.

Booter developers can refer to the cover function definitions, as they apply directly to the functions defined within the boot operations structure (`JBIBootOps`), with the exception of the `BootContextID` parameter.

Device tree navigation boot operations return opaque device tree entries (`JBIDevTreeEntry`).

All other boot operations return a 32-bit signed value (`long`); a value of less than zero indicates failure, otherwise the value is to be treated as returned data.

The C cover functions invoke the `BootOps` using variants on the following code snippet:

```
extern JBIBootOps *jbis;/* Saved by startJavaOS() */
extern JBICid cid;/* Saved by startJavaOS() */
/* Cover function boot operation XXX */
long jbiXXX(param1, parm2, param3)
{
return (jbis->jbiXXX)(JBICid cid, param1, param2, param3);
}
```

Function Summary

Function Category	Function Name
Miscellaneous	jbiGetVersion
	jbiQuiesce
	jbiDebugPrintf
	jbiGetPlatformID
	jbiGetMachineID
	jbiGetBootStatus
	jbiGetBootData
	jbiGetRebootVector
File Access	jbiGetFileList
	jbiGetFile
Memory Management	jbiGetPhysicalMemoryMap
	jbiGetVirtualMemoryMap
Device Tree Navigation	jbiDevTreeGetRootEntry
	jbiDevTreeGetBootEntry
	jbiDevTreeGetDebugEntry
	jbiDevTreeGetParentEntry
	jbiDevTreeGetFirstChildEntry
	jbiDevTreeGetPeerEntry
Device Tree Property	jbiDevTreeGetPropertyNameLength
	jbiDevTreeGetNextProperty
	jbiDevTreeGetPropertyValueLength
	jbiDevTreeGetPropertyValue
	jbiDevTreeSetPropertyValue
Extended	jbiGetFunctions

Miscellaneous Functions

jbiGetVersion

```
long jbiGetVersion(JBICid cid);
```

Parameters:

none

Returns:

`-1`: Failure

`>0`: The boot interface version number

Comments:

Get the boot interface version number. The boot version number is used by the caller to determine the boot interface version, so that it knows which functions are available from the booter and the calling syntax.

The initial set of functions is version 1 of the JBI. Any subsequent version is always a higher version number.

When new functions are added to the `BootOps` structure, the version number is incremented. Subsequent versions of the JBI are always super-sets of functionality relative to existing releases.

jbiQuiesce

```
long jbiQuiesce(JBICid cid);
```

Parameters:

none

Returns:

`-1`: Failure

`0`: Success

Comments:

Shuts down the booting system software. Shutting down causes the boot system to free memory allocated for booting, clean up its resources, and cede control to the Microkernel.

The `BootContextID` is invalid once this function returns, and the boot interface can no longer be called.

If the booting system is constructed upon a PROM such as OpenBoot, the booter should issue a quiesce call to the PROM at this time.

From the time the Microkernel is started until the quiesce call is made, the Microkernel avoids using any memory occupied by the booter. However, after the quiesce call is made, the Microkernel may choose to fully consume and re-map all of memory to suit the needs of JavaOS for Business.

jbiDebugPrintf

```
long jbiDebugPrintf(JBICid cid, const char *format, ...);
```

Parameters:

`Format`: Standard Posix printf format string

`...`: A variable length list of zero or more values to print to the booter debug device.The Microkernel assumes the booting system implements the printf using polled I/O.

Returns:

`-1`: Failure

`0`: Success

Comments:

Print a message using the debug device output stream. The debug device can be determined by calling:

`jbiDevTreeGetDebugEntry`

See Also:

`jbiDevTreeGetDebugEntry`

jbiGetPlatformID

```
unsigned char *jbiGetPlatformID(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure

`~NULL:`Success -> a C string IDs the platform.

Comments:

Returns a C string that identifies (IDs) the platform. A platform is a family of like machines such as the JavaStation. This string should match a platform name directory and JSD entry on the machine boot and user config servers respectively.

See Also:

`jbiGetMachineID`

jbiGetMachineID

```
unsigned char *jbiGetMachineID(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure

`~NULL:` Success -> a C string IDs this machine.

Comments:

Returns a C string that IDs the machine. A machine is a unique implementation of a platform. The machine may augment the platform by adding to the standard platform definition.

This string should match a machine name directory and JSD entry on the machine boot and user config servers respectively. A network adaptor's MAC address is typically used.

See Also:

`jbiGetPlatformID`

jbiGetBootStatus

```
JBIStatusCode jbiGetBootStatus(JBICid cid, JBIBootOpID opID, void
**data);
```

Parameters:

`opID:` The ID of the boot operation to return status. This ID is the same value as used to index into the boot operations table.

`data:` A pointer to data in the booter system that provides more status information. The structure for the memory is boot operation dependent.

Returns:

`-1:` Failure.

`>0:` A status code regarding the last `opID` operation.

Comments: Returns additional status information.

jbiGetBootData

```
void *jbiGetBootData(JBICid cid);
```

Parameters:

none

Returns:

`-1:` Failure.

`>0:` Success -> a pointer to a booter implementation-specific data structure.

Comments:

Returns a booter specific data structure. In this case, the Microkernel must be booter implementation aware.

See Also:

`jbiGetBootStatus`

jbiGetRebootVector

```
JBIReboot jbiGetRebootVector(JBICid cid,unsigned int *extended);
```

Parameters:

Extended: Pointer to memory where the extended support flag is to be stored. When the flag is true, the reboot format supports extended reboot functionality. Otherwise, the booter only provides a default (simple) reboot vector value as used to index into the boot operations table.

Returns:

`NULL:` No reboot vector available.

`>0:` Success -> a vector address. If the extended flag is true, this vector can be cast to the `JBIExtendedReboot` type.

Comments:

The memory containing the reboot vector must be in RAM or ROM that is marked as `rangeIsData`. This memory is consumed by the OS after the `jbiQuiesce` call.

The JBI vector prototype is:

```
void (*JBIReboot)();
```

The extended JBI vector prototype is:

```
int (*JBIExtendedReboot)(JBIRebootType type, void *data);
```

The list of extended reboot types is:

```
typedef enum {
rebootHard, /* Hardware reboot */
rebootSoft, /* Software reboot */
rebootDump, /* Dump memory and then reboot */
rebootCustom = 0x1000, /* Booter specific reboot codes
begin here */
} JBIRebootType;
```

See Also:

`jbiGetBootStatus`

File Access Functions

jbiGetFileList

JBIFileList *`jbiGetFileList(JBICid cid, JBIPhysMemoryMap **map);`

Parameters:

`map:` Address of the physical memory map structure allocated by the booter.

Returns:

`NULL:` Failure.

`>0:` Success -> a pointer to a `JBIFileList` structure.

Comments:

The booting system returns a pointer to a `JBIFileList` structure. This pointer is only valid up until the point of booter shutdown. Refer to `JBIQuiesce`.

The file name and data pointers within the structure however, remain valid past booter shutdown so these files may be appended to the ROM file system. Refer to `rangeIsData` type of Physical Memory range.

See Also:

`JBIFileList` structure definition.

jbiGetFile

`long jbiGetFile(JBICid cid, char *fileName, char *buffer, int size);`

Parameters:

`filename:` Pointer to a string allocated by the caller, in which the full path and name of the file is contained.

buffer: Pointer to memory allocated by the caller, where the contents of the file are loaded.

size: Maximum size of the buffer.

Returns:

`0:` Failure.

`>0:` Length of file.

Comments:

The client gets a file from the server by the same method used to get JavaOS for Business. The `fileName` is a pointer to the name of the file on the server and `buffer` is the JavaOS for Business memory where the data in the file is put. This function is not supported after `jbiQuiesce`.

Memory Management Functions

jbiGetPhysicalMemoryMap

```
long jbiGetPhysicalMemoryMap(JBICid cid, JBIPhysMemoryMap **map);
```

Parameters:

`map:` The address of the variable which, upon successful return, will contain the address of the physical memory map structure allocated by the booter.

Returns:

`-1:` Failure

`0:` Success

Comments:

The booting system returns a pointer to the physical memory map. Upon successful return, `map` is the address of the variable containing the address of the physical memory map structure allocated by the booter.

The physical memory map is a table of the raw, un-translated memory areas available on the platform. This map is the view of platform memory without the aid of an MMU. Most simple booters will have a statically defined table. Other more complex booting systems may build the table dynamically.

Any memory allocated by the booter for this table is released upon the `jbiQuiesce` call.

Refer to the Physical Memory section for details concerning the contents of the physical memory map.

See Also:

`jbiGetVirtualMemoryMap`

jbiGetVirtualMemoryMap

```
long jbiGetVirtualMemoryMap(JBICid cid, JBIVirtMemoryMap **map);
```

Parameters:

`map:` The address of the variable that, upon successfu return, will contain the address of the virtual memory map structure allocated by the booter.

Returns:

`-1:` Failure

`0:` Success

Comments:

The booting system returns a pointer to the virtual memory map. Upon successful return, `map` is the address of the variable containing the address of the virtual memory map structure allocated by the booter.

The virtual memory map is a table of the memory areas currently mapped on the platform.

Most simple booters will have a statically defined table. Other more complex booting systems may build the table dynamically.

Any memory allocated by the booter for this table is released upon the `jbiQuiesce` call.

Refer to the Virtual Memory section for details concerning the contents of the virtual memory map.

See Also:

`jbiGetPhysicalMemoryMap`

Device Tree Navigation Functions

jbiDevTreeGetRootEntry

```
JBIDevTreeEntry jbiDevTreeGetRootEntry(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure.

`>0:` Opaque ID of the root device tree entry.

Comments:

Obtain the entry for the root of the device tree. All other entries are children of the root.

See Also:

`jbiDevTreeGetBootEntry, jbiDevTreeGetDebugEntry`

jbiDevTreeGetBootEntry

```
JBIDevTreeEntry jbiDevTreeGetBootEntry(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure

`>0:` Opaque ID of the boot device tree entry.

Comments:

Obtain the entry for the boot device. This represents the device from which JavaOS for Business was booted.

See Also:

`jbiDevTreeGetDebugEntry, jbiDevTreeGetRootEntry`

jbiDevTreeGetDebugEntry

```
JBIDevTreeEntry jbiDevTreeGetDebugEntry(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure.

`>0:` Opaque ID of the debug device tree entry.

Comments:

Obtain the entry for the debug device. This device is the device to which bytes are written using `jbiDebugPrintf`, which is typically a video display or serial port.

See Also:

`jbiDevTreeGetBootEntry, jbiDevTreeGetRootEntry, jbiDebugPrintf`

jbiDevTreeGetParentEntry

```
JBIDevTreeEntry jbiDevTreeGetParentEntry(JBICid cid, JBIDevTreeEntry
entry);
```

Parameters:

entry: Device tree entry of the node for which the parent entry is requested.

Returns:

`NULL:` Failure, either `entry` was invalid or `entry` is the root, for which there is no parent.

`>0:` Opaque ID of the parent device tree entry.

Comments:

Get the parent of the specified entry in the device tree. Each entry in the device tree has a single parent entry, except for the root entry which has no parent. Each entry can have zero or more children.

See Also:

`jbiDevTreeGetRoot`

jbiDevTreeGetFirstChildEntry

```
JBIDevTreeEntry jbiDevTreeGetFirstChildEntry(JBICid cid,
JBIDevTreeEntry entry);
```

Parameters:

entry: Device tree entry of the node for which the first child entry is requested.

Returns:

`NULL:` Failure, either `entry` was invalid or `entry` has no children.

`>0:` Opaque ID of the first child device tree entry.

Comments:

Gets the first child of the specified entry in the device tree. The term *first* is arbitrary. All other children of `entry` are obtained as siblings of the first child using:

`jbiDevTreeGetPeerEntry.`

See Also:

`jbiDevTreeGetPeerEntry`

jbiDevTreeGetPeerEntry

```
JBIDevTreeEntry jbiDevTreeGetPeerEntry(JBICid cid,
JBIDevTreeEntry entry);
```

Parameters:

entry: Device tree entry of the node for which the peer entry is requested.

Returns:

`NULL:` Failure, either `entry` was invalid or `entry` has no further peers.

`>0:` Opaque ID of the sibling of `entry`.

Comments:

Gets the sibling of the device tree entry.

See Also:

`jbiDevTreeGetFirstChildEntry`

Device Tree Property Functions

jbiDevTreeGetPropertyNameLength

```
long jbiDevTreeGetProperyNameLength(JBICid cid);
```

Parameters:

none

Returns:

`-1:` Failure

`>0:` The maximum length for all property names in the device tree. Booters will typically return the value of `JBI_PROP_NAME_MAX`.

Comments:

All property names for all entries in the device tree have a maximum length. This function should be called *before* retrieving any property names so that the caller can allocate the proper amount of memory for the name. Booters typically return the value of `JBI_PROP_NAME_MAX`.

See Also:

`jbiDevTreeGetNextProperty`

jbiDevTreeGetNextProperty

```
long jbiDevTreeGetNextProperty(JBICid cid, JBIDeviceTreeEntry entry,
const char *prevPropName, char *nextPropName);
```

Parameters:

`entry:` Device tree entry ID to retrieve the property name.

`prevPropName:` Pointer to a string, allocated by the caller, in which the name of the entry's previous property is contained. Used a pointer value of `NULL` to get the first property of the entry.

`nextPropName:` Pointer to a string, allocated by the caller, in which the name of the next property is returned.

Returns:

`-1:` Failure, `entry` or `prevPropName` is invalid, or there are no more properties for this device tree entry.

`0:` No more properties exist for this device tree entry.

`1:` Success.

Comments:

Get the next property of the designated entry.

See Also:

`jbiDevTreeGetPropertyNameLength, jbiDevTreeGetPropertyValue`

jbiDevTreeGetPropertyValueLength

```
long jbiDevTreeGetPropertyValueLength(JBICid cid, JBIDevTreeEntry
entry, const char *propName);
```

Parameters:

`entry:` Device tree entry ID to retrieve the property value length.

`propName:` Pointer to a string, allocated by the caller, in which the name of the entry's property is contained.

Returns:

`-1:` Failure, `entry` or `propName` is invalid.

`0:` There is no value associated with the property. Zero length property values are used with boolean properties, where the presence or absence of the property indicates state.

`>0`: The length of the property value in bytes.

Comments:

Get the length of a property value of the designated entry. This function should be called *before* getting a property value to determine how much memory to allocate for the value.

See Also:

`jbiDevTreeGetPropertyValue, jbiDevTreeSetPropertyValue`

jbiDevTreeGetPropertyValue

```
long jbiDevTreeGetPropertyValue(JBICid cid, JBIDevTreeEntry entry,
const char *propName, void *value, long length);
```

Parameters:

`entry:` Device tree entry ID to retrieve the property value.

`propName:` Pointer to a string, allocated by the caller, in which the name of the entry's property is contained.

`value:` Pointer to memory, allocated by the caller, to which the property value is written.

`length:` Length, in bytes, of memory pointed to by `value`.

Returns:

`-1:` Failure, `entry` or `propName` is invalid.

`0:` There property has no value (boolean).

`>0:` Number of bytes copied to memory at `value`.

Comments:

Get the value of the named property of the designated entry. The caller must allocate the memory to hold the property value. The proper buffer size can be obtained by calling `jbiDevTreeGetPropertyValueLength`.

See Also:

`jbiDevTreeGetPropertyValueLength, jbiDevTreeSetPropertyValue`

jbiDevTreeSetPropertyValue

```
long bootOpsDevTreeSetPropertyValue(JBICid cid, JBIDevTreeEntry
entry, const char *propName, void *value, long length);
```

Parameters:

`entry:` Device tree entry ID to set the property value.

`propName:` Pointer to a string, allocated by the caller, in which the name of the entry's property is contained.

`value:` Pointer to memory, allocated by the caller, which contains the new value for property `propName`.

`length:` Length, in bytes, of memory pointed to by `value`.

Returns:

`-1:` Failure, `entry` or `propName` is invalid or this boot operation is not supported.

`>=0:` Number of bytes copied from memory at `value`.

Comments:

Set the value of the named property of the designated entry. Setting property values is optional. Some simple booting systems will not support this feature.

See Also:

`jbiDevTreeGetPropertyValue, jbiDevTreeGetPropertyValueLength`

Extended Functions

jbiGetFunctions

```
jbiExtFunctionTable *jbiGetExtFunctions(JBICid cid);
```

Parameters:

none

Returns:

`NULL:` Failure.

`>0:` Pointer to `jbiExtFunctionTable`.

Comments:

This function returns any extended functions that are supported by the client. This returns a pointer to the structure `jbiFunctionTable` if successful, else `-1`;

If the Function ID is negative, then the function is Vendor Specific. Also 1024 chunks of function ID's for NCs, Web Phones, PDAs, etc., can be reserved.

This function is not supported after `JBIQuiesce` and would be a base JBI function.

See Also:

`jbiExtFunctionTable; jbiExtFunction structure`

Standard Device Properties

Introduction

The device tree hierarchy, and its properties, is the means by which platform device information is communicated. However, the amount of information provided by the booter and discovered by JavaOS for Business itself can vary from one platform to another.

The manner in which the device information is conveyed can include:

- A full OpenBoot (IEEE 1275-1994) implementation within the booter that discovers all devices and provides this information to JavaOS for Business, which in turn, refers to the device tree for all device information.
- A booter that provides minimal configuration information to JavaOS for Business (perhaps only some platform type ID), and JavaOS for Business discovers any pertinent devices, building its own device database.
- A combination of the above.

The actual discovery of devices can be implemented as:

- Probing and identifying hardware devices.
- A hard-coded data structure for fixed configuration systems.
- A combination of the above.

Note that static configuration data can be maintained within the booter executable itself, within the JavaOS for Business platform and bus manager classes, flash memory, other local media, or even the boot server.

The JBI dictates mechanisms by which information is conveyed from the booter to JavaOS for Business. The physical and virtual memory maps are specifically defined, but the device tree contents are not.

The JBI defines how the device tree is navigated and how property names and values are obtained by JavaOS for Business, but it does not precisely dictate the device tree hierarchy and what property names and values are expected.

This flexibility allows for small embedded systems to avoid the overhead associated with a full IEEE 1275 device tree implementation, while permitting OpenBoot based systems to provide the maximum amount of information via the JBI.

Too much flexibility, on the other hand, can result in the non-portability of JavaOS for Business components. Using the JavaOS Device Interface (JDI) and JavaOS Platform Interface (JPI) device drivers are thought of as bus-specific and platform-independent.

The platform-dependent pieces include the booter. This is why there is so much flexibility in determining where the device discovery duties reside.

However, before the platform manager is started under JavaOS for Business, using the JBI, the device tree (as represented by the booter) is copied into the JavaOS for Business Object Database for use by the platform manager.

General Property Characteristics

A property is associated with a given entry in the device tree. It consists of a name and a value. An entry must have, at a minimum, a property with the name of *name* whose value is a null-terminated string that names the entry.

The property name is an ASCII string. The allowable characters are those defined by the IEEE 1275-1994 standard:

- Name consists of one to thirty-one printable characters.
- Names shall not contain uppercase characters, or the characters
  -)
  - \
  - :
  - [,]
  - @
- Property names beginning with the character + are reserved for use by future revisions of the IEEE 1275 standard.

The property value is simply an array of zero or more bytes. A zero-length property value is typically used when a given property represents a boolean state with its presence (true) or absence (false).

Otherwise, the value can represent a:

- sequence of bytes
- null-terminated text string
- 32-bit integer

or

- composite of any of these types

Integer values are represented in *big-endian* format. That is, bytes with greater numerical significance appear at lower memory addresses.

Byte swapping is required on some platforms. The byte and text string values are a linear sequence of bytes. Composite values are a concatenation of other values, without padding or alignment.

Both the producer and consumer of properties must agree on the format (or encoding) of the value of a given property name. This involves the booter and the JavaOS for Business platform and bus managers.

The IEEE 1275-1994 standard and the related bus bindings documents define property name and value formats. These documents are identified as:

> IEEE Standard 1275-1994, *Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*, October 28, 1994, ISBN 1-55937-426-8.
>
> PCI Bus Binding to: IEEE Std 1275-1994, Revision 1.7, Unapproved Draft.
>
> ISA/EISA/ISA-PnP binding to: IEEE Std 1274-1994, Revision 0.2, April 12, 1996, DRAFT
>
> IEEE Draft Std P1275.2/D14a *Standard for Boot (Initialization Configuration) Firmware Supplement for IEEE 1496 (SBus) Bus*.

IEEE 1275-1994 can be ordered from IEEE by calling (800) 678-IEEE and ordering product number SH-17327-0-001994-1-0. Using anonymous ftp, the bus bindings documents can be found on `playground.sun.com` in `/pub/p1275/bindings`.

JBI Source Files

Introduction

This section lists the JBI header file (`jbi.h`) to be included by all booter and Microkernel code making references to the JBI types and function prototypes, as well as the Microkernel OS-independent implementation of the C cover functions defined within the this section.

The cover functions define the JBI interface for the rest of the Microkernel.

Header File

The `jbi.h` header file is listed in its entirety below. It defines all of the types and function prototypes used to define the interactions between the JBI booter and the Microkernel. The cover functions, used only within the Microkernel, are also defined at the end of the file.

This file is located in `src/kona/java/and includes`:

```
#if !defined(JBI_H)
#define JBI_H

/*
* JavaOS Boot Interface definition.  This file defines all data
* types and structures pertaining to the JBI.  Refer to the JOSB
* Boot Interface Specification for a thorough overview of the
* items defined in this file.
*/

#ifdef __cplusplus
extern “C” {
#endif

/*
*  Current version of the JBI.  All versions are backwards
compatible.
*/
#define JBI_VERSION                         1L

/*
*  Physical memory map entry names.  Each entry in a physical
* memory map is of one of these types.
*/
```

```
typedef enum {
   rangeIsNotValid/* Range not usable (i.e. hole) */
   rangeIsROM,   /* Read-only or flash memory */
   rangeIsRAM,   /* Physical RAM */
   rangeIsFixedIO,   /* Memory map for permanent I/O devs */
   rangeIsExpansonIO,/* Memory map for dynamic I/O devs */
   rangeIsPlatformControl,/* Platform specific control */
   rangeIsCustom    /* Custom range name */
} JBIPhysMemoryRangeType;


/*
 *  The content of each physical memory range is defined here.
 */
typedef enum {
    rangeIsFree,         /* Free for JavaOS kernel use */
    rangeIsBooter,       /* In use by booter until aquiesced */
    rangeIsKernel,       /* Portion of JavaOS kernel loaded here */
} JBIPhysMemoryRangeContent;

/*
*  Location of a physical memory range.
 */
typedef struct {
    void *paddr;         /* Starting physical address */
    unsigned long length;/* Length of range in bytes */
} JBIPhysMemoryRange;

/*
* Physical memory map entry, supplies all information regarding a
 * specific range of physical memory.
 */
```

```
typedef struct {
                JBIPhysMemoryRangeType type;
                JBIPhysMemoryRangeContent content;
                JBIPhysMemoryRange range;
} JBIPhysMemoryMapEntry;

/*
 *  JBI Physical memory map structure.  This struct is variable
length.
 */
typedef struct {
   unsigned long numEntries;        /* Number of entries in map */
   JBIPhysMemoryMapEntry entry[1];  /* Contiguous array of map
entries */
} JBIPhysMemoryMap;

/*
 *  Virtual memory range caching attribute.
 */
typedef enum {
   rangeIsUncached,
   rangeIsCachedWriteThrough,
   rangeIsCachedWriteBack
} JBIVirtMemoryRangeCaching;

/*
*  Location of a mapped virtual memory range.  For a given range,
 * both the virtual and physical addresses are contiguous.  A
 * virtually contiguous area of memory mapped to physically
 * discontiguous pages results in multiple ranges being defined.
 */
```

```
typedef struct {
   void *vaddr;                /* Starting virtual address */
   void *paddr;                /* Starting physical address */
   unsigned long length;       /* Length of range in bytes */
} JBIVirtMemoryRange;

/*
 *  Virtual memory map entry, supplies all information regarding a
 *  specific range of virtual memory.
*/
typedef struct {
   JBIVirtMemoryRangeCaching caching;
   JBIVirtMemoryRange range;
}JBIVirtMemoryMapEntry;

/*
 *  JBI Virtual memory map structure.  This struct is variable
length.
 */
typedef struct {
   unsigned long numEntries;       /* Number of entries in map */
   JBIPhysMemoryMapEntry entry[1]; /* Contiguous array of map
entries */
} JBIVirtMemoryMap;

/*
 *  JBI (OpenBoot/IEEE 1275) device tree definitions.
 */
#define JBI_PROP_NAME_MAX       32L /* Max property name length
(w/NULL) */
typedef void * JBIDevTreeEntry;     /* Device tree entry type */

typedef long JBICid;                /* Boot Context ID type */
```

```
typedef enum {
   rangeIsUncached,
   rangeIsCachedWriteThrough,
   rangeIsCachedWriteBack
} JBIVirtMemoryRangeCaching;

/*
*  Location of a mapped virtual memory range.  For a given range,
both
 *  the virtual and physical addresses are contiguous.  A virtually
 *  contiguous area of memory mapped to physically discontiguous
pages
 *  results in multiple ranges being defined.
 */
typedef struct {
   void *vaddr;                /* Starting virtual address */
   void *paddr;                /* Starting physical address */
   unsigned long length;       /* Length of range in bytes */
} JBIVirtMemoryRange;

/*
 *  Virtual memory map entry, supplies all information regarding a
 *  specific range of virtual memory.
 */
typedef struct {
    unsigned long numEntries;   /* Number of entries in map */
    JBIVirtMemoryMapEntry entry[1];/* Contiguous array of map
entries */
} JBIVirtMemoryMap;

/*
 *  JBI (OpenBoot/IEEE 1275) device tree definitions.
 */
#define JBI_PROP_NAME_MAX       32L /* Max property name length
(w/NULL) */
```

```
typedef void * JBIDevTreeEntry;     /* Device tree entry type */

typedef long JBICid;                /* Boot Context ID type */

/*
 *  The following structure contains function prototypes defining
 * the complete set boot operation functions provided by the
 * booting system.  The address of this structure is provided as
 * the second argument when the JavaOS for Business entry point is
invoked by the booter.
 *
*
 * The opaque booter context ID (CID) is always the first parameter
 * provided by the caller.
 */
typedef struct {

    long (*bopGetVersion)(JBICid cid);
    long (*bopQuiesce)(JBICid cid);
    long (*bopDebugPrintf)(JBICid cid, const char *format, ...);

    long (*jbiGetFile)(JBICid cid, char *fileName, char *buffer,
int size);

    long (*jbiGetFile)(JBICid cid, char *fileName, char *buffer,
int size);
    jbiExtFunctionTable *(*jbiGetExtFunctions)(JBICid cid);
    JBIFileList (*jbiGetFileList)(JBICid cid);
    JBIStatusCode (*jbiGetBootStatus)(JBICid cid, JBIBootOpID
opid,
    void **data);
    unsigned char *(*jbiGetPlatformID)(JBICid cid);
    unsigned char *(*jbiGetMachineD)(JBICid cid);
    void *(*jbiGetBootData)(JBICid cid);
```

```
    JBIReboot (*jbiGetRebootVector)(JBICid cid,unsigned int
    *extended);

/*
*  Obtain memory type and usage information.
*/
          long (*jbiGetPhysicalMemoryMap)(JBICid cid,
          JBIPhysMemoryMap **map);
          long (*jbiGetVirtualMemoryMap)(JBICid cid, JBIVirtMemory
          Map **map);

/*
*  Navigate the device tree.  These functions return a handle
*  to the requested device tree entry.
*/
JBIDevTreeEntry (*jbiDevTreeGetRootEntry)(JBICid cid);
JBIDevTreeEntry (*jbiDevTreeGetBootEntry)(JBICid cid);
JBIDevTreeEntry (*jbiDevTreeGetDebugEntry)(JBICid cid);
JBIDevTreeEntry (*jbiDevTreeGetParentEntry)(JBICid cid,
  JBIDevTreeEntry entry);
JBIDevTreeEntry (*jbiDevTreeGetFirstChildEntry)(JBICid cid,
  JBIDevTreeEntry entry);
JBIDevTreeEntry (*jbiDevTreeGetPeerEntry)(JBICid cid,
  JBIDevTreeEntry entry);

/*
*  Obtain property/value information on a device tree entry.
*/
long (*jbiDevTreeGetNameLength)(JBICid cid);
long (*jbiDevTreeGetNextProperty)(JBICid cid, JBIDevTreeEntry
entry,
   const char *prevPropName, char *nextPropName);
long (*jbiDevTreeGetValueLength)(JBICid cid, JBIDevTreeEntry
entry,

    long (*jbiGetFile)(JBICid cid, char *fileName, char *buffer,
```

```
   const char *propName);
long (*jbiDevTreeGetPropertyValue)(JBICid cid, JBIDevTreeEntry
   entry,const char *propName, void *value, long length);
long (*jbiDevTreeSetPropertyValue)(JBICid cid, JBIDevTreeEntry
entry,
   const char *propName, const void *value, long length);

} JBIBootOps;

/*
 *  Entry point symbol name and type in JOSB executable to be
invoked
 *  by the booter.
 */
#define JBI_OS_ENTRY                “startJavaOS”
typedef (*JBIStartOS)(JBICid cid, JBIBootOps *jbijbis, void
   stackAddr,unsigned long stackLength);

/*
 *  Cover function prototypes.  These functions are invoked within
the
 * Microkernel and hide the boot ops pointer dereference and
 * the use of the boot context ID from the rest of the Microkernel.
 * These functions simply invoke each equivalent boot operation
   function.
 */
extern long jbiGetVersion(void);
extern long jbiQuiesce(void);
extern long jbiDebugPrintf(const char *fmt, ...);

extern long jbiGetPhysicalMemoryMap(JBIPhysMemoryMap **map);
extern long jbiGetVirtualMemoryMap(JBIVirtMemoryMap **map);
```

```
extern JBIDevTreeEntry jbiDevTreeGetRootEntry(void);
extern JBIDevTreeEntry jbiDevTreeGetBootEntry(void);
extern JBIDevTreeEntry jbiDevTreeGetDebugEntry(void);
extern JBIDevTreeEntry jbiDevTreeGetParentEntry(JBIDevTreeEntry
entry);
extern JBIDevTreeEntry
jbiDevTreeGetFirstChildEntry(JBIDevTreeEntry entry);
extern JBIDevTreeEntry jbiDevTreeGetPeerEntry(JBIDevTreeEntry
entry);

extern long jbiDevTreeGetProperyNameLength(void);
extern long jbiDevTreeGetNextProperty(JBIDevTreeEntry entry,
 const char *prevPropName, char *nextPropName);
extern long jbiDevTreeGetPropertyValueLength(JBIDevTreeEntry
entry,
 const char *propName);
extern long jbiDevTreeGetPropertyValue(JBIDevTreeEntry entry,
 const char *propName, void *value, long length);
extern long jbiDevTreeSetPropertyValue(JBIDevTreeEntry entry,
 const char *propName, const void *value, long length);

#ifdef __cplusplus
}
#endif
#endif /* JBI_H */
```

C Cover Functions

In the OS independent portion of the source tree, the C cover functions are defined. These functions invoke the equivalent boot operations using the parameters provided by the booter to the `startJavaOS` entry point. These parameters are the opaque `BootContextID` and the pointer to the booter's boot operations structure (`JBIBootOps`).

All cover functions are defined in `src/kona/java/os/jbi.c`. These functions are invoked by other portions of the Microkernel, or by native methods used by the JavaOS for Business JDK Runtime.

Version 1.4 of `jbi.c` is listed in its entirety below.

```
#include <stdarg.h>
#include "jbi.h"

/*
 * The JBI cover functions are implemented in this file.  These
 * functions hide the details regarding the boot operations
pointer
 * and the contextID from the rest of the Microkernel.  Each cover
 * function simply invokes the equivalent boot operation function
   defined by the booter.
 */

/*
 *  Parameters provided by booter to startJavaOS() are saved here.
 */
JBICid                  JbiBootCid;
JBIBootOps              *Jbijbis;


long
jbiGetVersion(void)
{
  return (Jbijbis->jbiGetVersion)(JbiBootCid);
}


long
jbiQuiesce(void)
{

   return (Jbijbis->jbiQuiesce)(JbiBootCid);
```

```
}


long
jbiDebugPrintf(const char *fmt, ...)
{
    va_list ap;
    long rval;

    va_start(ap, fmt);
    rval = (Jbijbis->jbiDebugPrintf)(JbiBootCid, fmt, ap);
    va_end(ap);
    return rval;
}


long
jbiGetPhysicalMemoryMap(JBIPhysMemoryMap **map)
{
    return (Jbijbis->jbiGetPhysicalMemoryMap)(JbiBootCid, map);
}


long
jbiGetVirtualMemoryMap(JBIVirtMemoryMap **map)
{
     return (Jbijbis->jbiGetVirtualMemoryMap)(JbiBootCid, map);
}


JBIDevTreeEntry
jbiDevTreeGetRootEntry(void)
{
```

```
    return (Jbijbis->jbiDevTreeGetRootEntry)(JbiBootCid);
}


JBIDevTreeEntry
jbiDevTreeGetBootEntry(void)
{
    return (Jbijbis->jbiDevTreeGetBootEntry)(JbiBootCid);
}


JBIDevTreeEntry
jbiDevTreeGetDebugEntry(void)
{
    return (Jbijbis->jbiDevTreeGetDebugEntry)(JbiBootCid);
}


JBIDevTreeEntry
jbiDevTreeGetParentEntry(JBIDevTreeEntry entry)
{
    return (Jbijbis->jbiDevTreeGetParentEntry)(JbiBootCid,
entry);
}


JBIDevTreeEntry
jbiDevTreeGetFirstChildEntry(JBIDevTreeEntry entry)
{
   return (Jbijbis->jbiDevTreeGetFirstChildEntry)(JbiBootCid,
   entry);
}
```

```
JBIDevTreeEntry
jbiDevTreeGetPeerEntry(JBIDevTreeEntry entry)
{
    return (Jbijbis->jbiDevTreeGetPeerEntry)(JbiBootCid, entry);
}


long
jbiDevTreeGetNameLength(void)
{
    return (Jbijbis->jbiDevTreeGetNameLength)(JbiBootCid);
}


long
jbiDevTreeGetNextProperty(JBIDevTreeEntry entry, const char
*prevPropName,
 char *nextPropName)
{
    return (Jbijbis->jbiDevTreeGetNextProperty)(JbiBootCid, entry,
      prevPropName, nextPropName);
}


long
jbiDevTreeGetPropertyValueLength(JBIDevTreeEntry entry,
 const char *propName)
{
    return (Jbijbis->jbiDevTreeGetValueLength)(JbiBootCid, entry,
     propName);
}


long
```

```
jbiDevTreeGetPropertyValue(JBIDevTreeEntry entry,
 const char *propName, void *value, long length)
{
    return (Jbijbis->jbiDevTreeGetPropertyValue)(JbiBootCid,
           entry,propName, value, length);
```

Table of Contents

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V