JavaOS™ for Business™

JavaOS™ for Business™ Version 2.0

Reference Manual

June 1998

“Table of Contents” on page vii

COPYRIGHT

Preface

The JavaOS™ for Business™ Reference Manual describes the features, layers and interfaces of JavaOS for Business, emphasizing the linkages and overlaps of the layers and interface functionality. JavaOS for Business is the result of a joint ventuire between Sun Microsystems, Inc. and IBM.

This manual describes *what* the JavaOS for Business interfaces are, rather than providing the details of *how* the interfaces are implemented.

A low-level description of the Java operating system, which includes many details transparent to the end user, is available in other documents including the *JavaOS Porting Guide.*

Who Should Use This Book

The audience for this book includes Java application developers, device driver writers, OEMs, and system administrators.

Chapter Summary

Chapter 1, Introduction to JavaOS for Business, is an overview and a brief description of the JavaOS for Business Client-side OS, Client-Server model, JavaOS for Business layers, components and interfaces.

Chapter 2, JavaOS Device Interface (JDI). Description of the JavaOS JDI device interface and its components and interfaces.

Chapter 3, JavaOS Platform Interface (JPI). Discusses the JavaOS JPI Platform and its components.

Chapter 4, JavaOS System Database (JSD). Describes the JavaOS JSD database layer, components and interfaces.

Chapter 5, JavaOS Event System. Discusses the JavaOS Event Manager components.

Chapter 6, JavaOS System Loader (JSL). Provides an overview of the JavaOS JSL.

Chapter 7, JavaOS Boot Interface (JBI). Describes the JavaOS JBI and Booter.

Chapter 8, JavaOS Hosting classes, is an overview of the JavaOS for Business Hosting classes and their interaction with the JDK Runtime layer.

Chapter 9, JavaOS Extension Class Files, is an overview of the JavaOS for Business system extension class files.

Details on the public JavaOS for Business classes are available in a separate file, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed on-line and is free.

Related Publications

- JavaOS System Design Document 1.1 - IBM
- Java System Database version 1.04 - SunSoft
- JavaBeans API Specification version 1.01 - Sun MicroSystems
- Java In A Nutshell; 2nd Edition - OReilly
- Hotjava Users Guide
- The Java Virtual Machine Specification - Lindholm & Yellin, Addison Wesley

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/smi.ssoftpress/catalog/javaseries.html`.

Acknowledgments

We would like to thank the following writers for the source specifications of this reference manual:

JavaOS for Network Computers Feature Specification for the Gemini Release, Ron Karim, *Appendix* by Shripad Patki. This specification documents the Gemini Release discussing the addition of a portable device driver architecture and enhanced NC configuration management to the release of JavaOS (Luna 1.1).

Java Device Interface (JDI) Review Specification, Tom Saulpaugh, describes the device driver layer of JavaOS.

Java System Database Server, Bernard Traversat and Steve Woodward. This document provides detail on the server schema, client/server protocol and persistence related APIs and implementation specific to the network computer reference architecture.

Java System Database Server Specification I and II, Bernard Traversat and Tom Saulpaugh discusses the Java System Database (JSD) Server describing the hierarchical schema and the APIs used by client applications to access the database. The intent of the JSD server document is to specify the server side extensions of the Java System Database.

Java System Database Specification, Jeff Schmidt and Tom Saulpaugh, with Bernard Traversat and Tom Clements, discusses the Java System Database.

JavaOS Platform Interface Review Specification, Tom Saulpaugh, discusses the platform interface of JavaOS.

JavaOS System Events (JSE) Specification, Tom Saulpaugh, describes the JavaOS package of events.

Java Service Loader (JSL) Specification, Mike Sullivan and Jon Wagner, discusses the loading of java classes.

JavaOS Configuration Tool (JCT) Specification, Muschett, IBM, discusses the system application used to administer and configure software for JavaOS Network Computers.

JavaOS Interrupts Specification, Tom Saulpaugh, describes the JavaOS Interrupt package of classes.

JavaOS Memory and DMA Classes Specification, Greg Slaughter, describes the JavaOS Memory and DMA package of classes.

Java System Services Specification, Tom Saulpaugh, describes the Java System Service package of classes.

JavaOS Boot Interface Specification, Tom Saulpaugh, describes the JavaOS Boot Interface.

List of Figures

1

Introduction

This chapter introduces the Reference Manual for JavaOS™ for Business™, Version 2.0 as the result of a joint venture between Sun Microsystems, Inc. and IBM. In this manual, JavaOS for Business software layers, components, and interfaces are described including their relationships and linkages.

JavaOS for Business is a Java-centric operating system with a feature set and architecture designed to provide a complete Java environment for a wide range of products including network computers.

JavaOS for Business is constructed as a layered architecture. Each layer is designed to be removable and updated independently, meaning that the operating system can be divided into components arranged in combinations that provide for the unique needs of a product.

JavaOS for Business eliminates the need for a host operating system to run Java applications.

Network computers offer low-cost computing platforms with powerful server-side system administration. JavaOS for Business provides this environment with integrated networking support, eliminating the need for client administration such as data backups and new software updates. This significantly reduces the total cost of ownership (TCO) for enterprise users.

The JavaOS for Business layered architecture consists of:

- Stand-alone JDK Runtime environment
- Device drivers
- JavaOS Platform Interface
- Microkernel and memory architecture

Support for the full Java application programmers interface (API)

The JavaOS layered architecture is divided into platform-specific and platform-independent code.

The *platform-specific* code is compiled to native code and contains the:

- Microkernel
- JavaOS Virtual Machine (JVM)

The *Microkernel* supports the following:

- booting
- interrupt handling
- multiple threads
- traps and DMA handling

enabling users to run multiple applets at the same time or download information while running a Java application.

The *JavaOS Virtual Machine* (JVM) supports the Java:

- bytecode interpreter loop
- execution handling
- memory management
- threads
- class loading
- bytecode verifier

JavaOS for Business extends the memory model to optimize for systems with limited memory.

The *platform-independent* portion of the system contains the:

- JavaOS Device Drivers
- JavaOS Network Classes
- JavaOS Window and Graphics systems

The *JavaOS Device Drivers* are written in Java and are portable and extensible.

The *JavaOS Network Classes* include industry standard networking protocols such as TCP/IP, UDP and ICMP for basic transport and routing. Both DNS and INS are used for looking up host names and supplying user names and passwords during log-in.

JavaOS for Business supports both Reverse ARP and DHCP for discovering network addresses and eliminating client administration. System clients can access files on an NFS server and be managed using SNMP.

The *JavaOS Window system* controls all drawing to the screen, implements user interface components such as buttons, menus and scrollbars and handles management of overlapping windows.

The *JavaOS Graphics system* supports all common graphics calls, including draw, fill, lines, arcs and polygons and font rendering. Both the graphics and window subsystems support the Java *Abstract Windowing Toolkit* (AWT) in a memory-efficient way.

This chapter introduces the following:

- JavaOS for Business
  - Client-Side
  - Client-Server
  - Java Development Kit (JDK)
- JDK Runtime Layer
  - JDK Hosting Classes
  - JavaOS Events System
  - JavaOS System Database (JSD)
  - JavaOS Device Interface (JDI)
  - JavaOS Platform Interfaces (JPI)
- Microkernel
- JavaOS Boot Interface (JBI)
- JavaOS Booter and the PC Booter (x86) platform

JavaOS for Business

JavaOS for Business focuses on key services and standard interfaces for the Client-Side OS and the Client-Server OS.

Client-Side OS

As shown in Figure 1-1, the JavaOS Platform Interface (JPI) presents a Device Driver Kit (DDK) for developing loadable drivers in Java (JDI).

Since Java limits access to memory, the JavaOS Platform Interface provides secure memory via a Platform Manager framework for writing Device Drivers in Java.

The Java OS Platform Interface (JPI) of the system provides the following:

- JavaOS System Database (JSD), manages configuration information that describes what devices are present, what system software services are installed, what user and group attributes have been selected, and any application-specific information required.
- JavaOS System Loader (JSL), a stand-alone service loading mechanism
- JavaOS Event System to provide the framework necessary to identify, map and manage device drivers, network protocols, and file systems to be loaded into the client platform
- JavaOS Configuration Tool (JCT), manages the hardware platform configuration

JavaOS Platform Services (JSD, JSL, and Event System) can be configured as a stand-alone system on a separate machine for client services and heavy access balancing. The JavaOS Platform Services include the server component with configuration management and client facilities.

The JSD supports a client/server model that facilitates the implementation of a thin client accessing configuration information, services, applications, and user data from one or more servers on the network. One server may provide the client the executable and loadable services required to boot the client. Another may authenticate user login and provide user-specific configuration information to the client. Still others may be sources of application and user data.

In addition, the system provides a JavaOS Device Driver development Kit (JDDK) and a Java Application Development Kit (JADK) to allow licensees and vendors to develop portable device drivers and applications through the JavaOS Device Driver Interface (JDI).

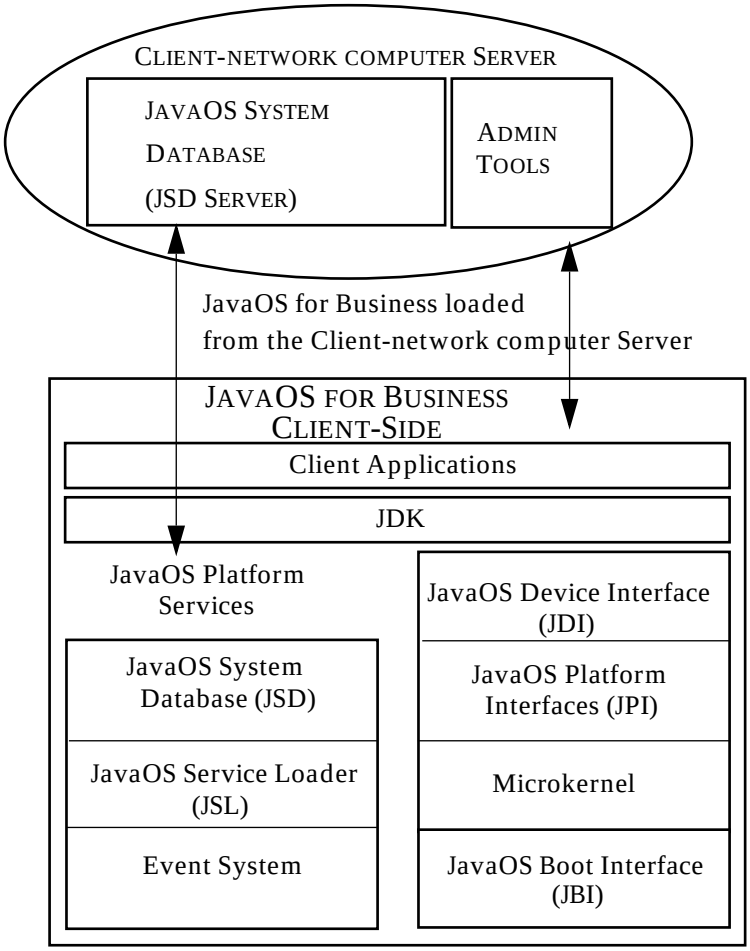


FIGURE 1-1 JavaOS for Business Client-network computers Server Overview

Client-Server Model

The main goal of the client-server computing model is to divide tasks. The user-interface is concentrated on the client side and the data management is performed on the server side. Figure 1-2 illustrates the Client-Server model.

Network computers are inexpensive client devices that interact with more powerful, centrally-located server computers. Java applets and applications run on these network computers and execute inside a single Java container application like the HotJava browser. These applets communicate through a network with applications running on a server.

The system is flexible enough to be ported to different hardware platforms. The system includes sample implementations that act as a guide to developing JavaOS for Business-based products. This includes the x86-based PC platform.

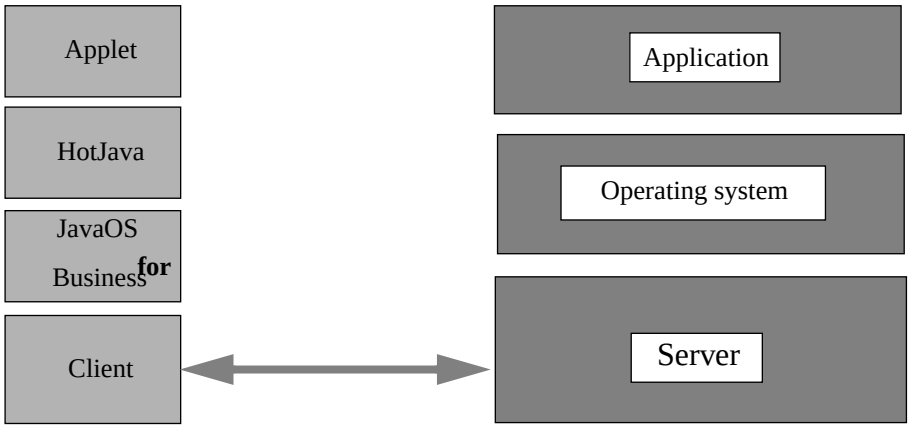


FIGURE 1-2 JavaOS for Business Client-Server Model

Some system features include:

- *File system is server-side.* Network computers do not have a local file system. The advantage of this is simplicity and lower cost. Without a local file system to back up, there is less risk of data loss.
- *Single programming language.* Other operating systems support many different programming languages. JavaOS for Business was designed to execute Java bytecode in a Java virtual machine.
- *No system calls.* Legacy operating systems provide access to operating system functionality through special interfaces. This programming style conflicts with the goals of 100% Pure Java because it creates software that is limited to specific environments.

Conventional PC's require a system administrator to be physically present for many support tasks. Since resources like fonts and system files are stored locally, the system administrator must be present to support each machine.

JavaOS for Business was designed to simplify this support model so that both the resources and the tasks to support those resources are concentrated on a server. This increases the productivity of a system administrator and lowers the cost of maintenance per seat.

JavaOS for Business is an operating system optimized to run Java applications in a network computing environment. It eliminates the need of a host operating system for running Java applications.

The system provides the following features:

- Stand-alone JDK Runtime environment
- Device driver architecture
- Portable device driver architecture
- Microkernel
- Configuration management

The device driver architecture leverages a collection of JPI components including the:

- Configuration database, the JavaOS System Database (JSD)
- Service loader, the JavaOS System Loader (JSL) capable of loading device drivers and other OS/JDK components
- Event Management system

Figure 1-3 presents an overview of the architecture layers containing the following components and interfaces:

- Client Application
- JavaOS Development Kit (JDK)
- JavaOS Virtual Machine (JVM)
- Hosting Classes
- JavaOS Device Interface (JDI)
- JavaOS Platform Interface (JPI)
- Microkernel
- JavaOS Boot Interface (JBI)

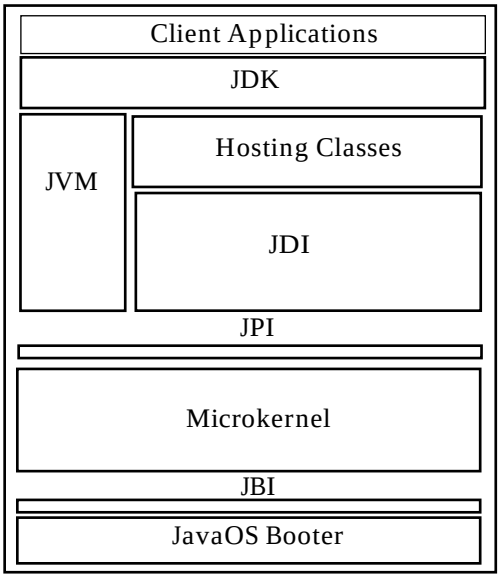


FIGURE 1-3 JavaOS for Business Layered Architecture

JDK and JDK Runtime Layer

As illustrated in Figure 1-4 the JDK and JDK Runtime Layer matches or maps to the JavaOS for Business. This section includes the following components:

- Java Development Kit (JDK)
- Java JDK Runtime layer including:
  - Java Virtual Machine (JVM)
  - JDK Hosting Classes
  - JavaOS Device Interface (JDI)
  - JavaOS Platform Interface (JPI)

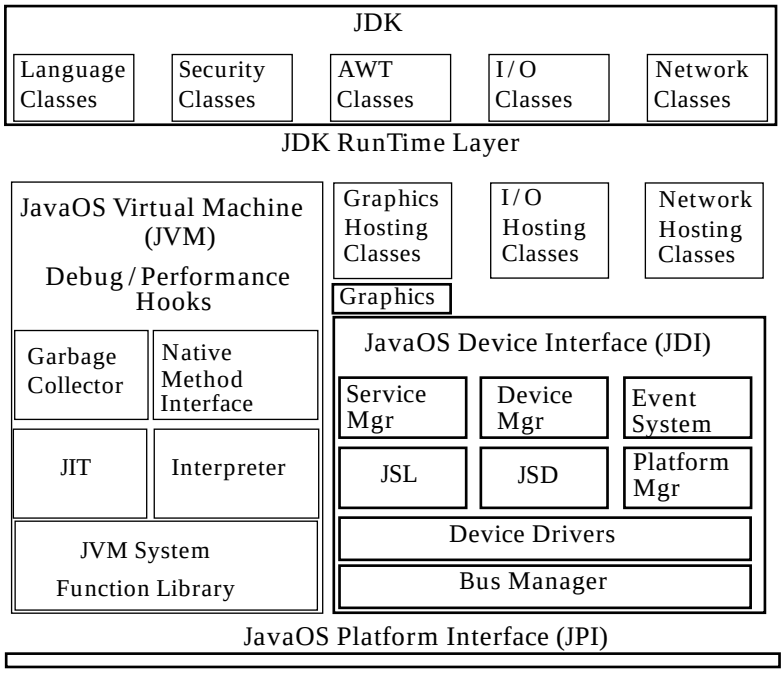


FIGURE 1-4 JDK and JDK Runtime Layers

Java Development Kit (JDK) Layer

The Java Development Kit (JDK) layer provides the means and the support for system layers and components through the use of the following classes:

- Language
- Security
- AWT
- I/O
- Network

Obtaining the name of a class or creating a new instance of a class requires that the developer derive an instance of class `Class` representing the current class. In turn, this requires that the developer know the name of the class or already have an instance of the class.

The Java platform includes a special class, the `Class` class, allowing the program to inspect Java data types at *JDK Runtime*. Each instance of class `Class` represents a different class or interface type loaded into the current Java Virtual Machine.

Using JDK, the developer queries a `Class` instance including the name of the class, class loader, superclass, interfaces, class fields, methods and constructors. JDK also allows the developer to create a new instance of the class and to access class objects directly from a class `Class` field.

For Events, JDK carries information about the event sources, the event type, when and where the event occurred, the state of the keyboard for the event, and the state of the component affected by the event.

JDK Runtime Layer

The layers above the Microkernel begin the platform-independent portion of the operating system and are collectively called the JavaOS for Business *JDK Runtime layer*. JDK Runtime components include:

- Java Virtual Machine (JVM)
- Code to support AWT
- Networking classes
- File-related I/O classes

JDK Runtime also includes the JDI or JavaOS Device Interface, which supports the device drivers written in Java. These drivers are an important client of the JavaOS Platform Interface or JPI.

The JDK Runtime contains support for AWT and I/O, as well for a modern device driver architecture. The AWT and I/O support modules use the JDI to perform I/O with a variety of devices.

In JavaOS for Business, AWT is layered upon a window system occupying space in the JDK Runtime layer. See Figure 1-5.

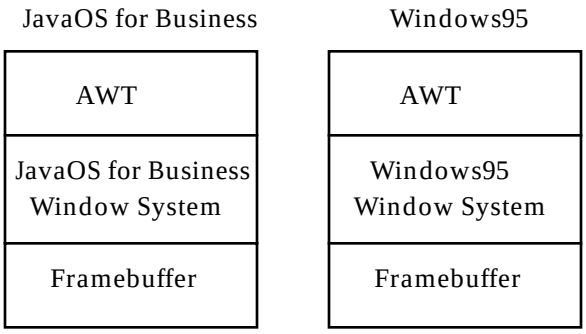


FIGURE 1-5 JDK Runtime Layer Middleware

The system is built from a combination of native code, some of which is platform-dependent, and Java byte-code, which is completely independent of the platform. The platform-independent portion of the operating system is the JDK Runtime, replacing the functions that reside within a host operating system supporting Java.

The JDK Runtime is also designed to run on a very hardware-limited platform. For example, the JDK Runtime does not require a Memory Management Unit (MMU) to map virtual addresses to physical memory addresses, or memory protection.

JDK Runtime allows developers to determine whether the underlying Microkernel should use an MMU or enforce memory protection. For example, the Microkernel bundled in the system uses an MMU to make several noncontiguous memory regions appear contiguous to that version of JDK Runtime.

The features implemented in this version of the system by the underlying Microkernel are shaped by the product under construction. For instance, an implementation destined for a phone needs a small, simple kernel. However, a network computer may require a more complex kernel implementation.

The JDK Runtime also lets the Microkernel designer decide how to manage multiple CPUs: symmetrically using SMP or asymmetrically with ASMP. The JDK Runtime is fully re-entrant. Each JDK Runtime critical section is protected by Java language synchronization primitives, and is therefore fully capable of running on a multiple processing platform.

From the Microkernel's point of view, JavaOS for Business is a process residing in a single JavaOS address space. The JavaOS address space may or may not resemble the physical memory map in size or layout.

Any differences between virtual and physical memory are managed by the Microkernel. JDK Runtime never assumes the physical characteristics of memory; JPI service calls gather all knowledge of JavaOS address space.

Some of the sublayers within the JDK Runtime include required and optional components for given product and/or platform combinations.

Required components for all products on all platforms include the following:

- Configuration Manager (CM): the first class of Java code to be executed by the JVM. The JVM regards the CM as its main application, invoking the CM main method with the standard Java application start-up parameters.
- The CM:
  - parses start-up arguments
  - starts up the platform manager component by passing argument strings
  - starts up the list of required services and the list of option services described by the start-up parameters.
  - The Java prototype for CM's main method is:

```
public static void main(String ags[]) {}
```

- Platform Manager: the component responsible for initialization of platform-specific devices and other hardware resources. The Platform Manager is the CPU host-bus manager.
- Service Manager: a utility package of Java code that can locate, load, and start JDK Runtime services.
- Device Manager: the central logic for the JDI driver architecture.
- Virtual Machine: JVM is an abstract computing machine with an instruction set that uses various memory areas. Common practice is that a virtual machine implements a programming language and the Java Programming Language uses JVM as the implementing method.
- JDK Classes specific to JOSD
- JPI Classes: insulate JDI device drivers and JDI support code from platform specific issues regarding time, memory, and interrupt management.

Optional components depend on the following factors:

- JDK configuration including B-JDK (Business), E-JDK (Embedded), or P-JDK (Personal)
- The set of devices requiring drivers (e.g., Framebuffer, ethernet, mouse, keyboard, serial, etc.)
- Debugging support needed.

For instance, Figure 1-6 illustrates the components in a JDK Runtime for a network computer:

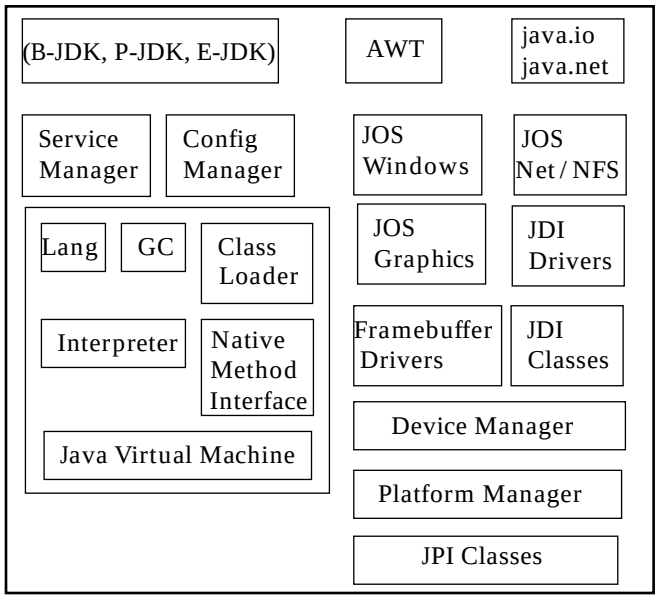


FIGURE 1-6 JavaStation network computer JDK Runtime Components

Components required for specific platforms and / or products:

- Platform-Independent Device Drivers: a key component of the JDK Runtime. All device drivers managing access to a platform resource of device use the JPI, making the system a platform-independent device driver model.

  However, platform independence doesn't mean bus independence. Each device driver is written to operate in the context of one or more expansion or I / O buses. Each driver uses the JPI to access platform resources such as memory and interrupts, but is also paired with a bus manager that allocates bus resources.

  Figure 1-7 shows that the model lets third-party expansion card vendors maintain a single copy of driver source code for each device and bus combination offered, resulting in substantial savings for the vendor.

  Each vendor must also use a JPI-compliant Microkernel, certified with a series of tests made available to each JavaOS for Business licensee.

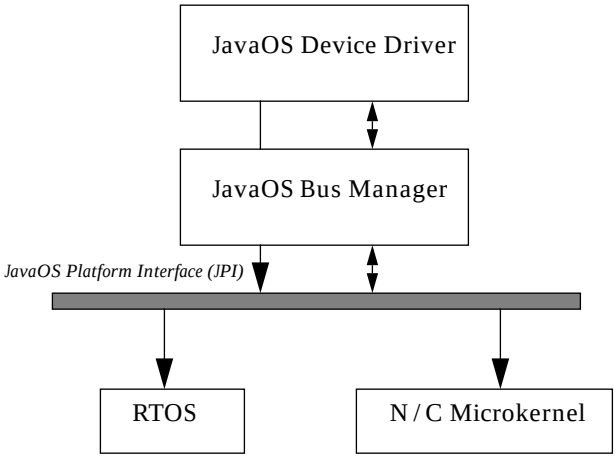


FIGURE 1-7 Device and Bus Management

- Platform-Tuned Device Drivers: While the JPI insulates drivers from platform dependencies, it doesn't prevent a driver from gathering platform-specific information that can optimize performance.
- A JavaOS for Business device driver can, if necessary, be written to be platform specific, since each component of the JPI publishes methods that return platform-specific information.

Java Virtual Machine (JVM)

The heart of the Java programming language is the Java Virtual Machine, (JVM), which is responsible for Java's:

- cross-platform delivery
- small footprint compiled code
- security capabilities

The JVM contained in the JDK Runtime layer of the system is an abstract computing machine with an instruction set that uses various memory areas. Common practice is that a virtual machine implements a programming language and the Java Programming Language uses JVM as the implementing method.

The JVM knows nothing of the Java programming language, only of a particular file form, the `class` file format. A `class` file contains JVM instructions (or bytecodes) and a symbol table, as well as other information.

The JVM components are as follows:

- Garbage Collector
- Native Method Interface
- JIT
- Interpreter
- JVM System Function Library

The JavaOS Virtual Machine (JVM) provided with the system has been optimized for the needs of network computers. This includes a just-in-time compiler (JIT), a memory management system optimized for limited memory environments and a Microkernel designed solely to support the needs of the JVM.

The development environment includes a tool for linking Java classes and generating multi-class files or preloaded sets of class files. These can be used to build application modules and more compact image files that can be stored in ROM.

Other Java JDK Runtime environments map abstract services (like Threads) required by the JVM onto services provided by the underlying native operating system (like native threads). The system provides these services directly.

Hosting Classes

The JDK Runtime layer includes Hosting Classes for:

- Graphics
- I/O
- Network

JavaOS Device Interface (JDI)

The JavaOS Device Interface (JDI) is a series of classes and interfaces providing appliance software access to *locally connected I/O devices.*

Clients of the JDI include applications, applets, JavaOS for Business system packages and JavaOS device drivers. Applets and various system packages use JDI to configure the I/O system and to initiate I/O requests. Drivers use the JDI to fulfill I/O requests.

The JDI functions as a device manager, managing a device through the first step in a device lifecycle called *discovery.* Device discovery is a platform, bus, and device-specific task.

The JDI calls upon the booting sub-system to publish devices found during system boot. The Bus management software alerts the JDI post-boot to the existence of a new device. PCI expansion buses allow for complete plug-and-play device management.

Once a device is discovered, it is *named* by the entity (JBI-PROM, bus manager or JSD administrator) that found the device. The name then serves as a *pointer* to the device.

The JDI includes the following components:

- Event System
- Platform Manager
- JavaOS System Database (JSD)
- Device Drivers
- Bus Manager

Figure 1-8 illustrates the JDI components.

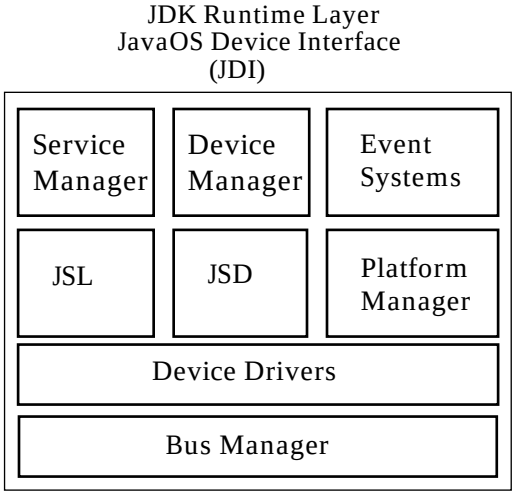


FIGURE 1-8 JDK Runtime Layer JDI Components

Event System

JavaOS for Business supports a local platform communication model based upon *Events*. An event is a typed piece of information.

The simplest event contains a single piece of information referencing the event source, known as the event *producer*.

The system event model supports multiple event receivers, called *consumers.*

Event, producer and consumer definition classes are provided a package of classes. This events package acts as the *broker* between event producers and consumers. The broker model of distribution serves to separate producers from consumers and lessen the burden on any one event producer.

Platform Manager

JDI Platform Management provides configuration naming services using these services to name devices and publish interfaces. JDI works with the JavaOS System Database (JSD).

JavaOS System Database (JSD)

JSD allows the storage and retrieval of Java System Configuration information through a unified repository on all Java platforms. The JSD lets operating system services, applications, JavaBeans, and JDK components store and retrieve complex configuration information about all Java platforms.

The database is populated dynamically during platform start-up using a flexible population interface. The population interface allows data to be obtained from files, the network, and the host OS. An interface to browse and navigate through the data hierarchy and contents is also provided.

The JSD can interact with other registries, like Windows NT™, and the data can be populated from the Windows NT registry. An Event mechanism is provided to maintain data coherency after the initial population of the database.

The server side of the JSD is a stand-alone system that exists on a separate server machine other than the JavaOS for Business server to better optimize high client access load.

The JSD uses a distributed repository model with the IIOP protocol for client and server communication.

Device Drivers

Device families and stand-alone device drivers are the building blocks of the sytem's I/O system. It is the collective set of families, drivers, and their relationships that implement the many I/O interfaces exported to device clients.

JDI device drivers provide families with a consistent, powerful loading and matching package that does most of the work of matching a driver to a device.

Bus Manager

The JDI Bus Manager ensures the driver is matched to the device for each bus connected to a platform. The Bus Manager allocates memory and process interrupts on behalf of the drivers

JavaOS Platform Interface (JPI)

JavaOS for Business extends the Java platform with portable, platform-independent interfaces to memory, interrupt, and I/O support classes. The Java classes themselves rely on the underlying Microkernel but provide the facilities for the Java Virtual Machine (JVM) and JDI device drivers to use platform-independent interfaces (insulated from platform dependencies.)

The JPI provides services to the virtual machine for threads, paging, and native code libraries. The OS extensions can use services like platform timing, interrupt and memory architecture.

JPI includes the following components:

- JVM System Function Library Managers
- Memory Classes
- Language Classes
- DMA Classes

Figure 1-9 illustrates the JPI components.

JavaOS Platform Interface (JPI)

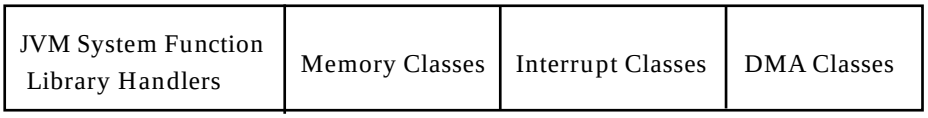


FIGURE 1-9 JPI Components

JPI clients are divided into two groups: native and Java. The primary *native* client is the Java Virtual Machine (*JVM*). The JVM views the platform interface as a set of C-based routines providing the raw platform services required by the virtual machine. This collection of services is called the *JVM System Functions.*

The Java-based clients include Java-based device drivers, and various system software components. A Java-based client views the platform interface as a set of support classes that are implemented as a combination of Java and native methods.

The JPI client composition is illustrated in Figure 1-10.

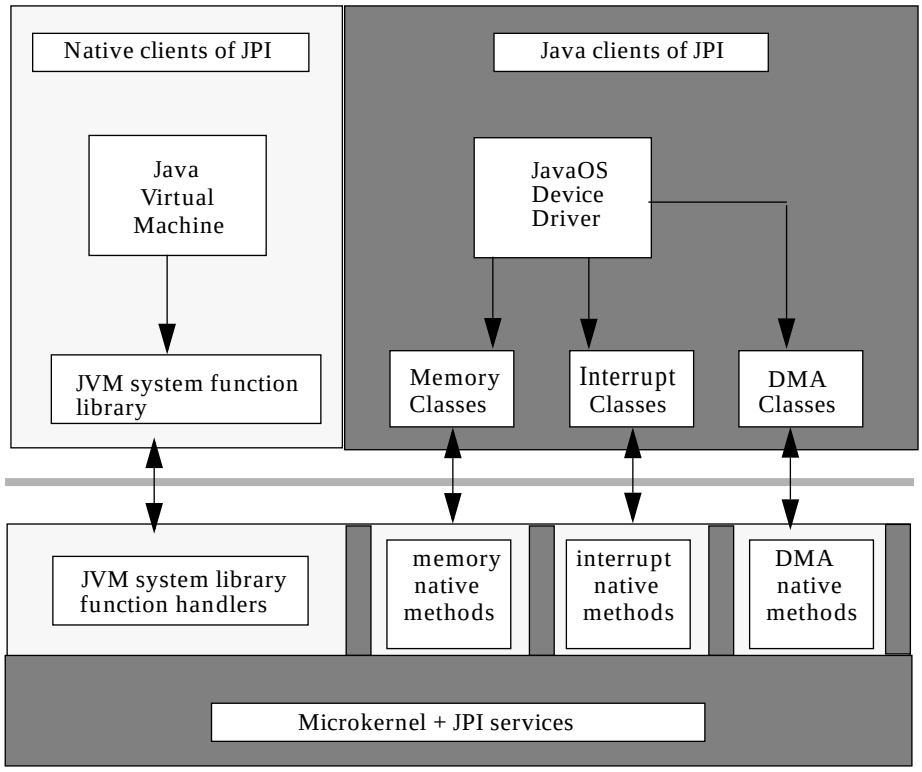


FIGURE 1-10 Clients of the Java Platform Interface

From the Microkernel point of view, the kernel provides native, C-based handlers that execute on behalf of the native and Java clients. The C-based handlers are *platform dependent*. The client-invocation side of the interface is *platform independent*.

The JPI is an interface designed to provide more flexibility between the Microkernel and JDK Runtime. It allows separate teams and/or companies to work on the Microkernel and JDK Runtime components of the system in parallel.

From a product point of view, one kernel could be tuned for hard realtime environments, while another kernel could be designed for a network computer.

The Microkernel was designed with an network computer in mind. It implements the JVM system functions and provides just enough device-related native methods to support linked drivers. The JDI has been created in part to support a formal, portable device driver model.

Microkernel

The platform-dependent part of the operating system is known as the Microkernel, a small component of native code that provides services to the virtual machine and to the OS extensions. Compliance with the JPI specification is validated by a suite of tests designed to exercise Microkernel services.

Services required of a JPI-compliant Microkernel are memory management, process management, thread management, and I/O support.

Microkernel components are illustrated in Figure 1-11:

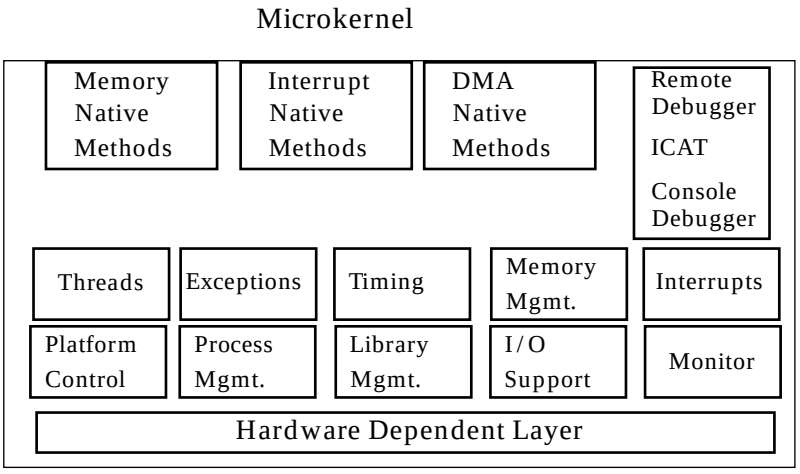


FIGURE 1-11 JavaOS Microkernel Layer

A JPI-compliant Microkernel performs four kinds of services:

- memory management
- process management
- thread management
- I/O support.

Memory Management

A JPI-compliant Microkernel exports a set of memory management services that let the JDK Runtime allocate virtual address space. A *page* is the smallest unit of memory managed by the Microkernel on behalf of the JDK Runtime. The JVM requires pages of virtual address space to build object heaps, but device drivers require pages of memory for buffer space.

Named collections of one or more pages of memory are called *areas*. All areas reside in the single JavaOS for Business address space, including space for the Microkernel itself. An area's name and type denote its purpose.

For example, an area of type RAM and named *JVM object heap* is used by the JDK Runtime to implement Java objects. Device drivers use an area of I/O space memory called *Framebuffer* to access I/O devices.

Figure 1-12 illustrates a sample virtual address space:

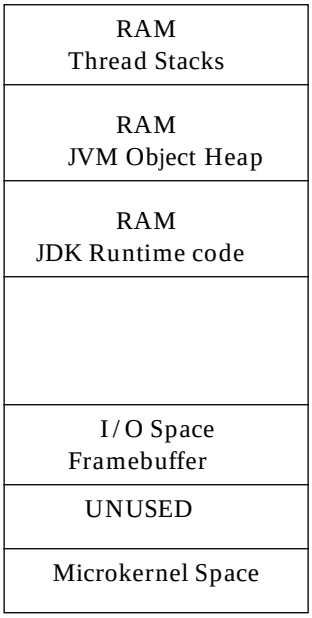


FIGURE 1-12 Virtual Address Space

Process Management

The Garbage Collector (GC) accounts for memory allocation within an object heap. Some resources required by the JDK Runtime, however, cannot be accounted for by the GC. Therefore, a JavaOS for Business process is considered the unit of Microkernel and JDK Runtime resource accounting. Processes are used by the JDK Runtime to track resources assigned to applications, JDK Runtime service modules, and device drivers.

Multiple processes share the JavaOS for Business address space. Each process is assigned a set of threads and memory areas.

The Microkernel exports a set of process management services that encapsulate the lifecycle of a process.A process is created without any assigned resources, but over its lifetime is dynamically assigned threads and memory areas. Process resources are consumed by the Microkernel when the process is deleted.

Thread Management

The Microkernel exports a set of thread management services that let the virtual machine fulfill the semantics of Java threads. Each Java thread is layered upon a Microkernel thread.

The Microkernel thread model supports multiple threads within a single JavaOS for Business process.

I/O Support

A JPI-compliant Microkernel exports a set of memory management services that let the JDK Runtime allocate virtual address space.

JBI (JavaOS Boot Interface)

Figure 1-13 illustrates the JBI and JavaOS Booter layers.

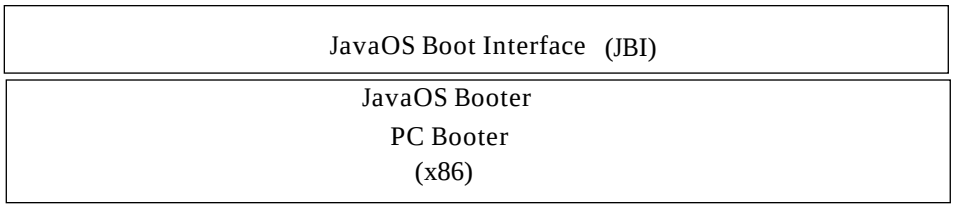


FIGURE 1-13 JavaOS Boot Interface and JavaOS Booter for the x86-based PC

The *booting layer* includes:

- JavaOS Boot Interface (JBI)
- JavaOS Booter

The booting system loads JavaOS for Business into memory and begins execution of the Microkernel. The system can be booted from network, ROM, RAM, CD-ROM, floppy, and hard disk devices.

The JBI standardizes the boot operations available to the operating system. This interface is used both by the Microkernel and the JDK Runtime layers.

JBI defines the bi-directional interface that is also an OS execution environment. JBI gives product designers flexibility when making booting decisions.

2

JavaOS Device Interface (JDI)

Introduction

This chapter describes the JavaOS for Business Device Interface (JDI) including the:

- JavaOS for Business Architecture Overview
- JDI and JavaOS for Business
- Device Management
- Device Handles
- JDI I/O Support Framework
- JDI I/O System Class Framework I/O Terminology
- Device Interface Hierarchy
- JDDK JavaOS Device Interface

The JavaOS for BusinessArchitecture Overview provides a quick illustration of the system layers including the Device Interface. The JDI and JavaOS for Business section discuss how JDI and the layered architecture of the system relate and illustrate the JDI components.

The next sections discuss Device Management and Device Handles providing details and an example of JDI use.

JDI I/O Support Framework and the JDI I/O System Class Framework I/O Terminology provide a details of the JDI I/O support framework and defines basic JDI concept terminology.

The Device Interface Hierarchy discusses in detail the device interface inheritance hierarchy of JDI.

The JDDK Device Interface describes the series of JDI device interfaces, exceptions, handles and driver classes for the Device Development Kit (DDK).

Architecture Overview

Figure 2-1 presents an overview of the architecture containing the following components and interfaces:

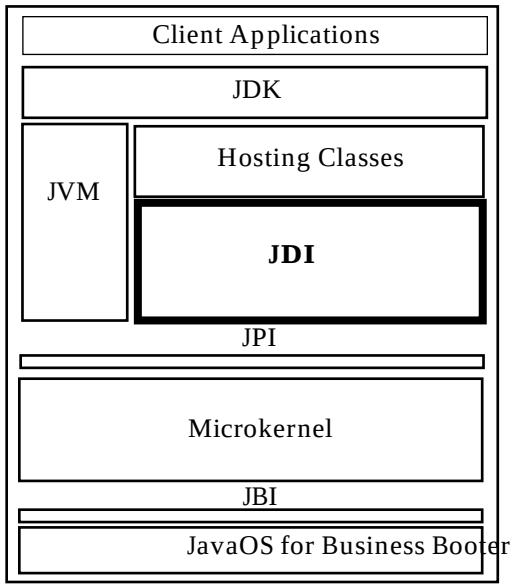


FIGURE 2-1 JavaOS for Business Layer Architecture

Included in the system layer architecture are the following:

- Client Application
- JavaOS Development Kit (JDK)
- JavaOS Virtual Machine (JVM)
- Hosting Classes
- JavaOS Device Interface (JDI)
- JavaOS Platform Interface (JPI)
- Microkernel
- JavaOS Boot Interface (JBI)

The JDI components within the architecture are shown in Figure 2-2.

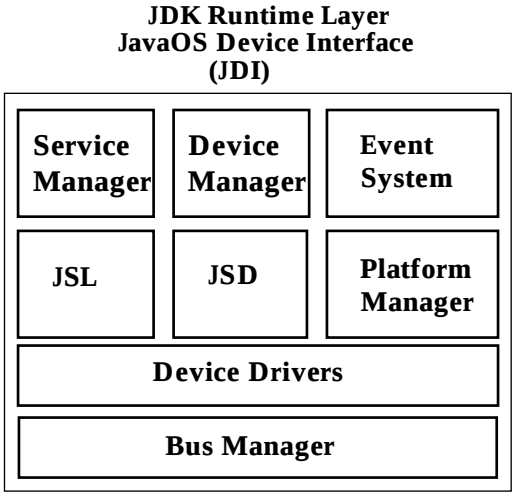


FIGURE 2-2 JDK Runtime Layer JDI Components

JDI and JavaOS for Business

JavaOS for Business is an operating system designed to run Java on a variety of low-cost computing hardware platforms without requiring a host operating system. The sole purpose of JavaOS for Business is to support the JavaOS Virtual Machine (JVM) and core Java APIs.

Other operating systems are complex and require significant hardware resources. Yet not all of these resources are necessary to support general purpose network computer applications on such platforms as thin-client desktop systems.

The system is simple to use and yet powerful and flexible enough to run systems with limited hardware resources. This simplicity makes system administration easier, therefore making it less costly to maintain a variety of platforms including desktop systems.

The system was designed to be as portable as possible, minimizing the hardware requirements necessary to support itself. Designed to support only JavaOS applications, the system can take advantage of Java language features. By limiting the hardware required to support JavaOS for Business as much as possible, new types of devices can be built that use the Java platform and take advantage of Java strengths.

Figure 2-3 illustrates where the JDI logically resides relative to the JDK APIs, JDK Runtime Layer, JVM, JavaOS Platform Interface and the Microkernel.

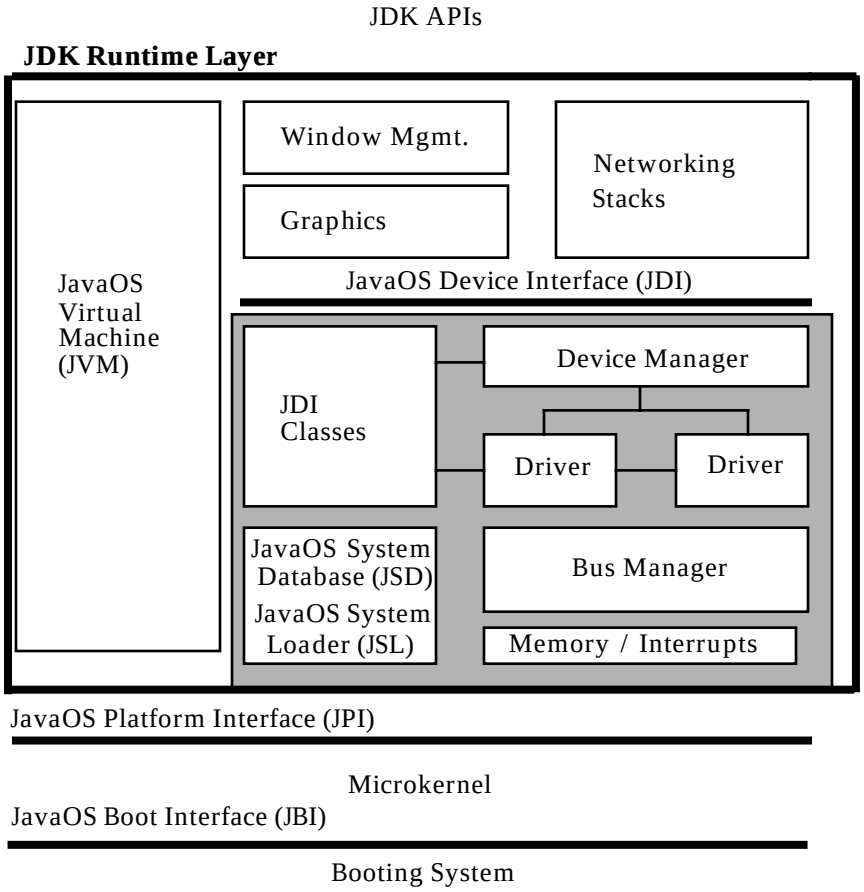


FIGURE 2-3 JDI Logical Placement

JavaOS for Business defines a platform to be:

- one or more CPUs
- a physical memory map
- any *permanently* attached devices, buses, and slots

The system platform-*independent* component is called the JavaOS *JDK Runtime layer.* (See figures 2-1 and 2-3).

The JDI resides in the JDK Runtime layer relying on the JPI to insulate the JDI and device drivers from platform-dependent issues.

The platform-*dependent* portion of the system is referred to as the *Microkernel layer.* (See figures 2-1 and 2-3).

The JDI is composed of 100% Java code. Drivers that are JDI-compliant run on many platforms with few or no source code changes required.

Device Management

The term Device Manager refers to the default device manager supplied by JavaOS for Business. JDI devotes much of its functionality to the management of devices. A device moves through a *lifecycle* from device discovery through garbage collection of the device object.

Discovery

The first step in the cycle is *discovery.* Discovery of devices takes place during the start-up phase of JavaOS and in post-boot as well.

Device discovery is a specific task for:

- platform
- bus
- device

The JDI calls upon the booting sub-system to publish devices found during system boot, and also allows bus management software to alert JDI post-boot to the existence of a new device.

Another option available to platform implementors is the *static definition of devices in the server JavaOS System Database (JSD).*

Modern expansion buses like PCIs allow for complete plug-and-play device management. Others, such as ISA require a static JSD definition.

The JDI imposes no rules upon the booting system regarding device discovery. The discovery can be completely *dynamic* or fully *static*. A PROM implementing the IEEE 1275 firmware standard can probe for devices each time the platform is booted, for example.

On the other hand, a small embedded device may have its set of physical devices fixed and not require any discovery code. The JDI only requires that the booting system publish its set of physical devices in the form of a device tree, *not that the tree must be built dynamically.*

Figure2-4 illustrates the device publication process:

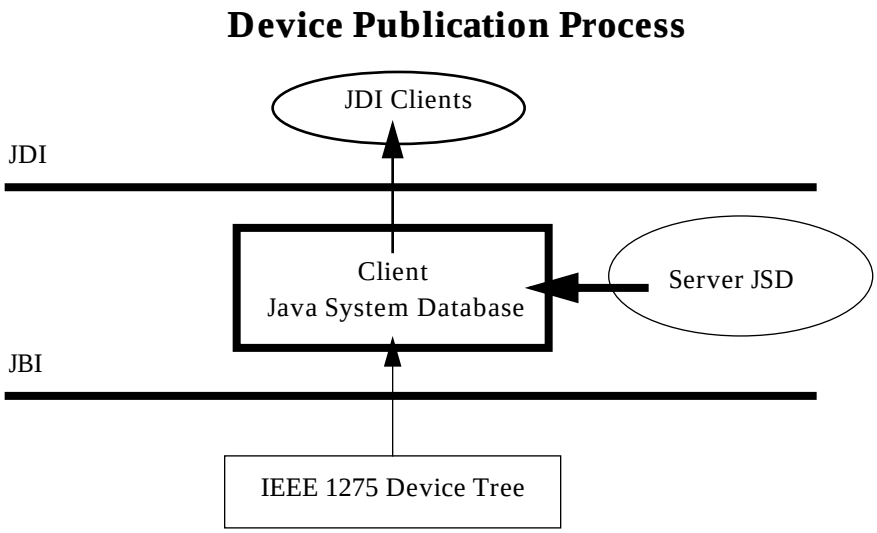


FIGURE 2-4 Device Publication Process

Naming

Once a device is discovered, it must be *named* by one of the following that discovered the device:

- JBI-PROM
- bus manager
- JSD administrator

Names serve as a public *pointer* to the device.

During the device publication process, the JDI may provide additional device name aliases as the device is added to the JSD. The naming of a device within the database also triggers a categorization process that may result in clients being alerted with an Event.

Devices begin life in an *off-line* state. Off-line devices do not have a driver associated with them and are therefore not able to participate in real I/O.

Device management classes in JDI serve to move the device from the off-line state to the on-line state by finding a driver for the device. Once a driver has been located for the device, the device can proceed to the *on-line* state.

On-line devices constantly move from idle to busy and back to idle again as the driver fields I/O requests from clients. At some point in time, a device might need to shutdown. The shutdown process causes the device to return to the off-line state. If the device is being removed from the system, its name entry is deleted as well.

Device Handles

An application constructs a *device handle* Java Bean.

Java Beans are a set of component APIs that are wholly written in Java. They exploit Java object-oriented reusability by letting developers create Java applets and applications from reusable components.

The device handle references the device driver (object) and a set of related action(s) or set of actions the device must perform.

All of the information about the interfaces, device handle, device driver (including any action(s) or set of related actions for a device) and the specific device (fax machine, printer, external hard drive, etc.), resides in the JSD.

The JSD stores and retrieves configuration information on behalf of applications and JDK system services like device drivers.

JavaOS for Business implements the device handle addressing the device driver, which in turn acts upon the specific device based on the information contained in the JSD. There can also be multiple devices (children of the first device in an hierarchical order) which is determined by the application as constructed by a client.

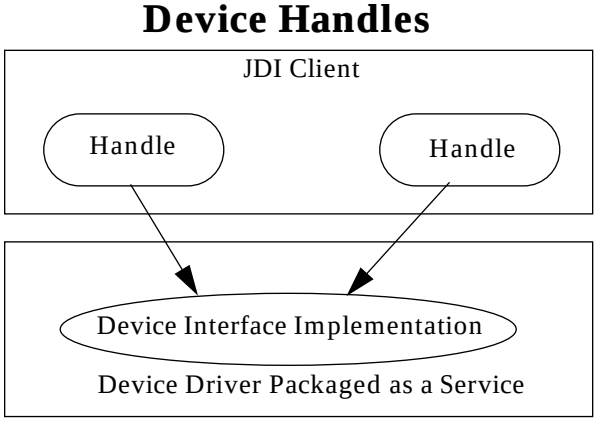


FIGURE 2-5 Device Handles and Drivers

Figure 2-6 illustrates an example of a device handle I/O connection to a fax machine.

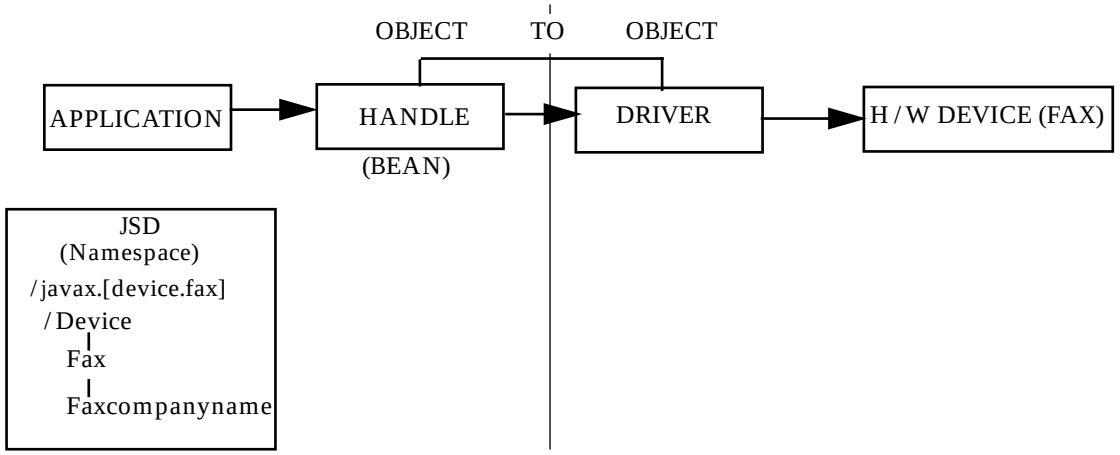


FIGURE 2-6 Device Handle Flow

Example:

Device handle I/O connection to a fax machine. The name of the interface, such as a package name for the fax machine, is:

```
javax.[device.fax.faxcompanyname]
```

Under this device handle name any action(s) or set of actions to be performed by the fax machine are listed, i.e. `send, receive,` etc.

JDI contains *namespace* information for a device, (for example, the fax machine), within the JavaOS System Database (JSD) component. The JavaOS System Loader (JSL) is the JDI component that automatically loads this information upon request.

The namespace is called the *device interface*. Device interface information for all devices is registered in the JSD.

JSL loads the device driver automatically in the namespace interface where the driver is registered.

Information strings are taken from the *service administration business card* profile and a pathname is created by changing:

`javax.[device.fax.faxcompanyname]`

`to`

`javax/[device/fax/faxcompanyname]`

using a slash (/) to replace each dot (.).

Example: `javax.[device.fax.faxcompanyname]`

Under this interface is a list of the operating actions, such as `Send` or `Receive`.

For example, a device handle to the fax machine contains a reference to the fax device (and related driver) and a set of fax actions like *send* (data from the fax) and *receive* (data to the fax) are defined as strings from the service advertisement.

For example, there are four steps taken to implement using a fax:

1. Create a device handle that points at a service advertisement.

   ```
   h = new FAXHANDLE
   ```

2. Open the device. JSL loads the driver. JSL binds the device to the device handle including the device action or set of actions. It also checks the exclusivity or sharing options for device use and sets the option.

   ```
   h.open (specific action or set of actions)
   ```

3. Implement the device action(s). Displays the driver loading, keeps the driver simple and address all device issues.

   ```
   h.action(s) or set of actions (byte[ ] array data)
   ```

4. Close or deactivate the device by unloading the device driver.

   ```
   h.close
   ```

All devices support device handles and a standard open and close interface that operates on a device handle.

Device handles begin life in a *closed* state and are marked as *open* when a device driver processes an open method. Figure 2-6 shows the device, device handle, and driver relationship.

NOTE: In some instances, developers of Point-of-Sale (POS) applications call device handles controls. For more information on this type of applications development see www.javapos.com.

JDI I/O Support Framework

All I/O-capable operating systems provide both mechanism and policy to ensure I/O occurs in a consistent, coherent fashion. JDI provides necessary mechanisms such as driver loading, and interface advertisement, as well as policy to ensure consistent I/O behavior across all types of devices.

JDI is a framework for supporting I/O. The JDI allows for unique device capabilities to be first class I/O interfaces.

Where possible, JDI defines interfaces that apply to all devices. In general, I/O housekeeping functions such as Open, and Close are normalized for all kinds of devices. Device events and naming are also areas where JDI presents unified mechanisms and policies.

JDI builds upon the strengths of the Java object-oriented programming model. The JDI defines a series of:

- Interfaces
- Abstract classes
- Concrete classes

that together form the I/O framework.

Clients always use an *abstract class* or *interface*. By forcing clients to use only abstract methods, the framework of concrete classes is freed to focus on implementation.

Role of Drivers under JDI

The role of a JDI compliant driver is to perform I/O.

Limiting drivers to just I/O allows the JDI to exist in embedded environments with limited resources (i.e. no display or keyboard).

Drivers needing to interact with the user must be paired with a standard applet or Java application. For example, some booting devices may need to ask the user for an encrypted password before proceeding with the JavaOS boot. Drivers communicate with an applet or application using the System Event subsystem.

Driver Code Re-use

Today, an unprecedented pace of device driver development is the result of the ever-increasing matrix of CPUs, devices, and buses. The ability to reuse some or all of an existing driver (to create a new one using the existing driver as a template) is of great value to developers.

Java device drivers are written using the Java programming language and are therefore CPU-independent. Driver developers must however, still manage the many device and bus combinations.

Some commercial operating systems *attempt to normalize all bus operations* on behalf of device drivers. When successful, normalized bus operations allow a driver for a device on *Bus A* to be built from the same code as when managing the same device on *Bus B.*

The price of normalized bus operations is:

- Complex drivers and complex I/O system components
- Bigger drivers and bigger I/O system components
- Greater testing burden
- Slower I/O

The system takes a different approach to the managing the matrix of devices and buses.

JavaOS for Business advocates systematic code re-use instead of normalized bus operations.

The Java programming language is object-oriented and are well-suited to code re-use.

Java device drivers and bus managers are built using standard base classes. These base classes act as development templates. As mentioned previously in this section, the templates provide driver developers with a head start on the coding process.

The re-use design philosophy results in more but smaller, simpler drivers.

I/O System Class Framework Concepts Terminology

This section provides a list of JDI terms for describing the most basic concepts of the I/O system class framework.

Client

A *client* is any Java class 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 hardware entity or virtual hardware entity

Physical hardware devices are tangible and normally attached to an I/O bus.

The JDI relies on the JavaOS System Database(JSD), the JavaOS Boot Interface (JBI), and the Java System Loader (JSL) 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.

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

Device Interface

A *device interface* is a public Java I/O interface focused at accessing a device. *A device interface is tailored toward the abstraction of a virtual or physical device*, such as:

- ethernet controller
- timer
- disk partition
- SCSI I/O bus

The device interface defines the *contract* between the device *client* and interface *implementors*. The contract includes the methods, data, and semantics implemented in the interface, including buffering and any asynchronous behavior attributed to the device.

The system provides a *standard device interface (Device)* that is sub-classed to define new kinds of devices. The standard (Device) interface is a set of methods that allow a client to *open*, *close*, and *query* the state of any device.

The standard device interface provides the generic, normalized view of all devices in the system. This generic view of devices is called the *generic device category.*

Each subsequently defined device interface defines a more specific *category* of device.

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 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 SCSI Device interface is defined as:

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

The Serial Device Interface is defined as:

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

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) provides a unified SCSI interface for all flavors of SCSI bus controller.

The term Device Manager refers to the default device manager supplied by the system.

The system 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.

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 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 {...}
```

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 device driver or a device 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

A *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.

Data Type Interface

A *data type* interface is a public Java-style I/O interface supported by a device driver and/or device family. The implementation of the interface is the joint responsibility of the family members (manager + drivers).

A data type interface is a set of classes and methods *tailored toward a particular type of data* such as:

- block-oriented data
- streaming data
- serial data
- audio data

This interface implies data management characteristics, not the physical characteristics of devices capable of storing, retrieving, or handling this type of data.

Some device families:

- are device-oriented
- are data type-oriented
- can publish both device and data-type oriented interfaces

A device managed by a device family that implements one or more data-type interfaces causes the device to be viewed in terms of its data.

For example, the file system partitions placed on a disk drive become block-oriented virtual devices conforming to the block storage data-type interface.

The data-type interface defines the *contract* between the data *client* and data interface *implementor.* The contract includes the methods, data, and semantics implemented in the interface, including buffering and any asynchronous behavior attributed to the management of the data.

Utility Interface or Class

A *utility interface* is a JavaOS for Business interface that is orthogonal to either device control or data management. A good example of such an interface is power management.

JavaOS for Business drivers providing power management implement the *Power* interface.

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

A utility class provides functionality similar in nature to the JDK (`java.util`) package. Examples of this utility include buffer management and timing services.

I/O Request

An *I/O request* is the encapsulation of the state and data defining a *single interface transaction.* The interface is one of the following:

- device
- data-type
- utility

The system supports two kinds of I/O request transactions for all kinds of interfaces. Figures 2-7 and 2-8 illustrate the two transaction types.

Synchronous transactions are encapsulated by the JVM as a series of Java method calls beginning at a device handle. The client thread invokes a method in an interface that in turns invokes more methods until the request is satisfied by one or more drivers.

A single client thread context is used to hold the state of the request from beginning to end.

Synchronous transactions are the default mechanism for initiating I/O, because they are the most natural style of Java programming.

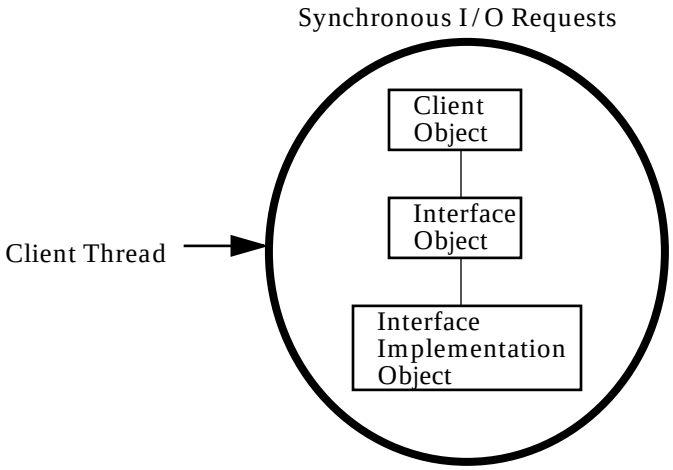


FIGURE 2-7 Synchronous I/O Requests

Conversely, an I/O that requires multiple threads to service cannot be handled using the standard sequential Java method invocation model. These kinds of I/O requests require an *asynchronous transaction*.

Asynchronous transactions are encapsulated as an *I/O request event object*. An I/O request event contains all the data and state required to define and maintain an I/O request *disassociated from any one thread.*

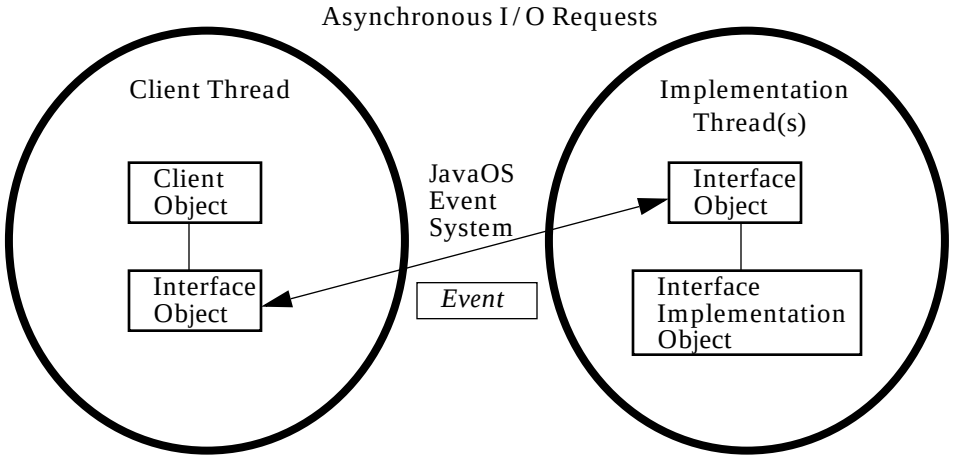


FIGURE 2-8 Asynchronous I/O Requests

Service

A service is a free-formed package of Java classes that can do nearly anything. A service has a few standard methods:

- Start
- Stop
- Suspend
- Resume

The rest of the package methods and classes are specific to the service provided. A service can be used to wrap existing APIs with a Java interface, for example.

A service does not have to be I/O related at all and can best be thought of as a typed library of Java code.

Clients, drivers, and managers are all implemented as services. The printing and networking subsystems make extensive use of JDI service packages to implement print drivers and network protocols.

Service Advertisement

A service advertisement is an object representing a public interface published in the namespace of the JSD interface.

Device handles are constructed using a service advertisement. Given the advertisement, a device handle can use the Java Service Loader (JSL) for an instance of the service (i.e., device driver).

The JSL automatically loads the service, if necessary.

Code Execution Context

Both managers and drivers are composed of Java classes. All code within a Java class executes relative to the currently scheduled thread.

A thread context is the smallest CPU specific set of resources needed to maintain a separately scheduled computing entity.

A driver or manager may be associated with one of three kinds of threads:

- *client*
- driver
- system

Client threads are threads created and managed outside the scope of the driver and JavaOS system packages. Examples of client threads include threads created and used by the HotJava browser.

Driver threads are threads created by the driver or manager itself. System threads are threads created and managed by JavaOS. Examples of system threads include the garbage collector, interrupt thread, and idle thread.

The *JavaOS for Business Event System* provides a standard mechanism for transferring control from one thread to another.

Address Space Terminology

An *address space* is a domain of related memory addresses. The JPI provides the complete memory architecture definition.

A list of some key memory address definitions are as follows:

Virtual Address Space

The domain of memory addresses that *may* be under the control of a translation mechanism such as an MMU and that can be referenced by the CPU in a natural programmatic fashion.

All JavaOS code executes in the context of the virtual address space.

All memory-mapped devices appear as a virtual address.

Virtual Memory

A range of virtual address space.

Physical Address Space

The domain of memory addresses NOT under control of a translation mechanism such as an MMU and that *cannot* be referenced from a software program executing on the CPU.

Physical Memory

A range of physical address space.

Device Interface Hierarchy

All devices are bound to one of more I/O interfaces. The set of interfaces for any one device is considered to be an *inheritance hierarchy*. There are two types of device interfaces:

- A base device interface defines a set of generic methods that all devices support.
- Custom device interfaces are created by *extending the device interface* or some other interface in the hierarchy, for example, Point-of-Sale (POS) Application devices.

The *base device* interface allows clients to get the device name or to retrieve other descriptive pieces of information such as lifecycle state, and which driver if any, is currently managing the device.

Once a device is under the management of a family and then a driver, its custom interfaces are implemented by a family member (manager and/or driver).

For example, a SCSI (Small Computer Systems Interface) bus controller device is managed by the SCSI family. As the family assigns a device driver to manage the bus, the device driver implements a SCSI device interface.

JDI Driver Classes

All JDI drivers are constructed from a *class hierarchy*. The most generic high-level class defines a set of methods such as:

- Start
- Stop
- Open
- Close ...

Drivers with custom methods are defined by extending an existing driver class in the hierarchy.

JDI Manager Interfaces

JDI manager interfaces are constructed from a *class hierarchy* the same as JDI drivers are. The most generic high-level class defines a set of methods such as:

- Probe
- Get Byte order
- Open
- Close ...

Some of these methods are passed on to the driver, while others are implemented within the manager.

Managers with custom methods are defined by sub-classing an existing manager class in the hierarchy.

Building Blocks: Families and Drivers

Device families and stand-alone device drivers are the building blocks of the I/O system. It is the collective set of families, drivers, and their relationships that actually implement the many I/O interfaces exported to device clients.

Device families can themselves be clients of other families. Each family publishes objects that implement a public I/O interface.

For example, the SCSI family manages all SCSI buses connected to the platform. For each connected bus, the SCSI family publishes an object that implements the standard JavaOS SCSI Bus Interface.

Further, for each bus the family makes sure a driver is matched to the device (in this case, the device is the SCSI bus controller). The JDI provides families a consistent, powerful loading and matching package that does most of the work of matching a driver to a device.

Figure 2-9 shows the generalized anatomy of any device family:

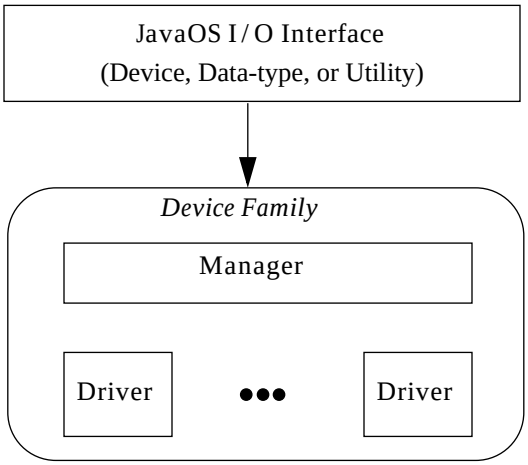


FIGURE 2-9 Anatomy of a Device Family

Room exists in all families for augmentation and customization. The JDI only imposes a framework and some standard public interfaces as family structural requirements.

An example of JDI implementation freedom is the use of native code by drivers in a family. A driver may be just a wrapper of an existing native driver exported by the JPI.

Families can also make use of loadable services.

The JavaOS network protocol family for example, is composed of a manager, and a series of protocols implemented as layered services. The network protocol manager loads and unloads protocols based upon client demand, maintaining a minimum memory footprint.

JDI Class Framework

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

Sun encourages additions to the JDI class framework to extend the reach and power of system. To make the construction of extensions easy for application developers, Sun provides base classes that act as *templates* for various kinds of drivers and managers.

Figure 2-10 shows how provided templates are leveraged into all components of the JDI framework.

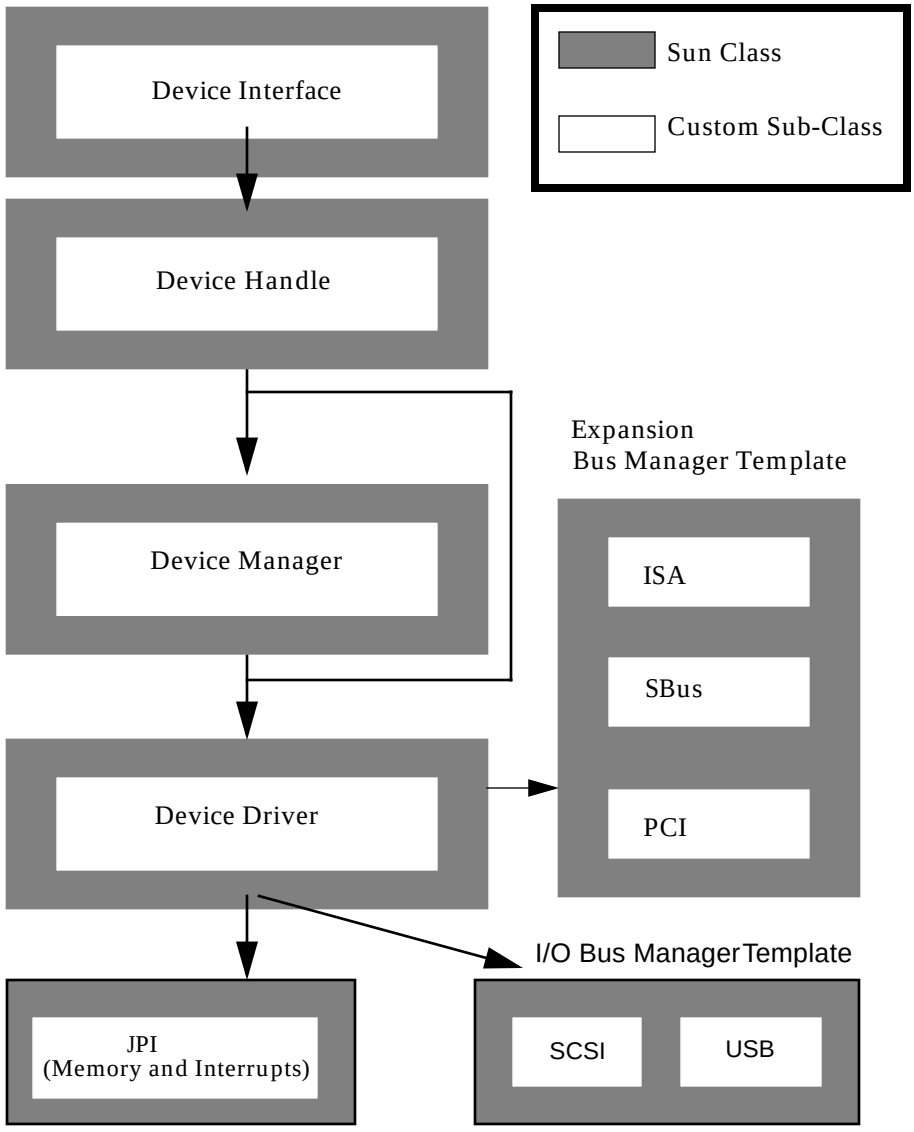


FIGURE 2-10 Template Use

Device Manager Class Hierarchy

As shown in Figure 2-11, the JDI framework of classes includes the following hierarchy of JDI device managers used as templates for creating new device managers:

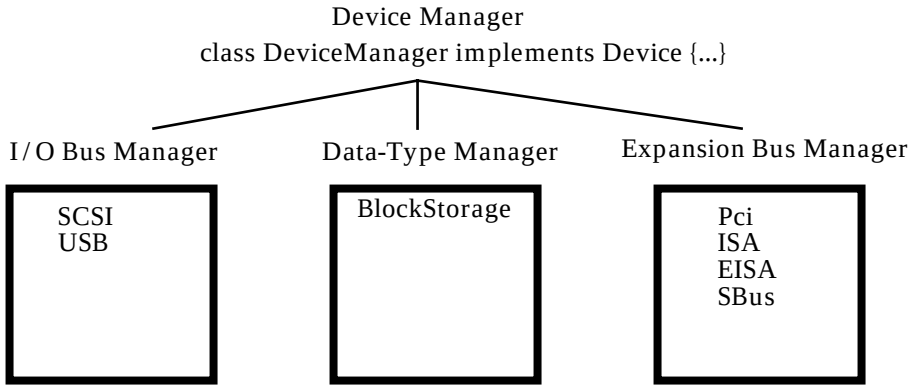


FIGURE 2-11 Manager Hierarchy Templates

Driver Class Hierarchy

In Figure 2-12, the JDI framework of classes includes the following hierarchy of drivers used as pre-formatted templates for creating new drivers:

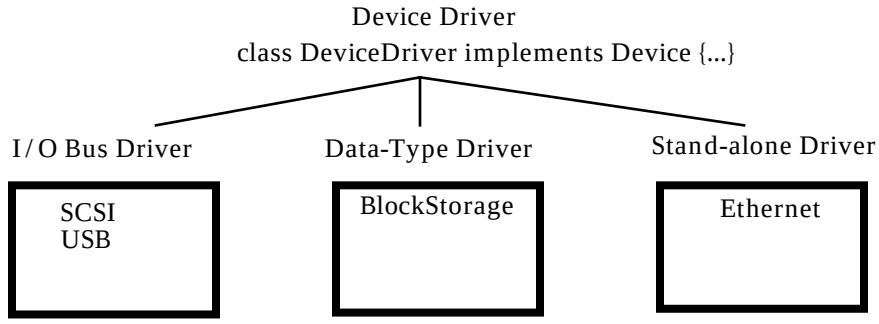


FIGURE 2-12 Driver Hierarchy Templates

Driver and Manager Layering

The JDI framework is used to construct many layers of cooperating managers and drivers. Figure 2-13 illustrates some typical layers within the JDI.

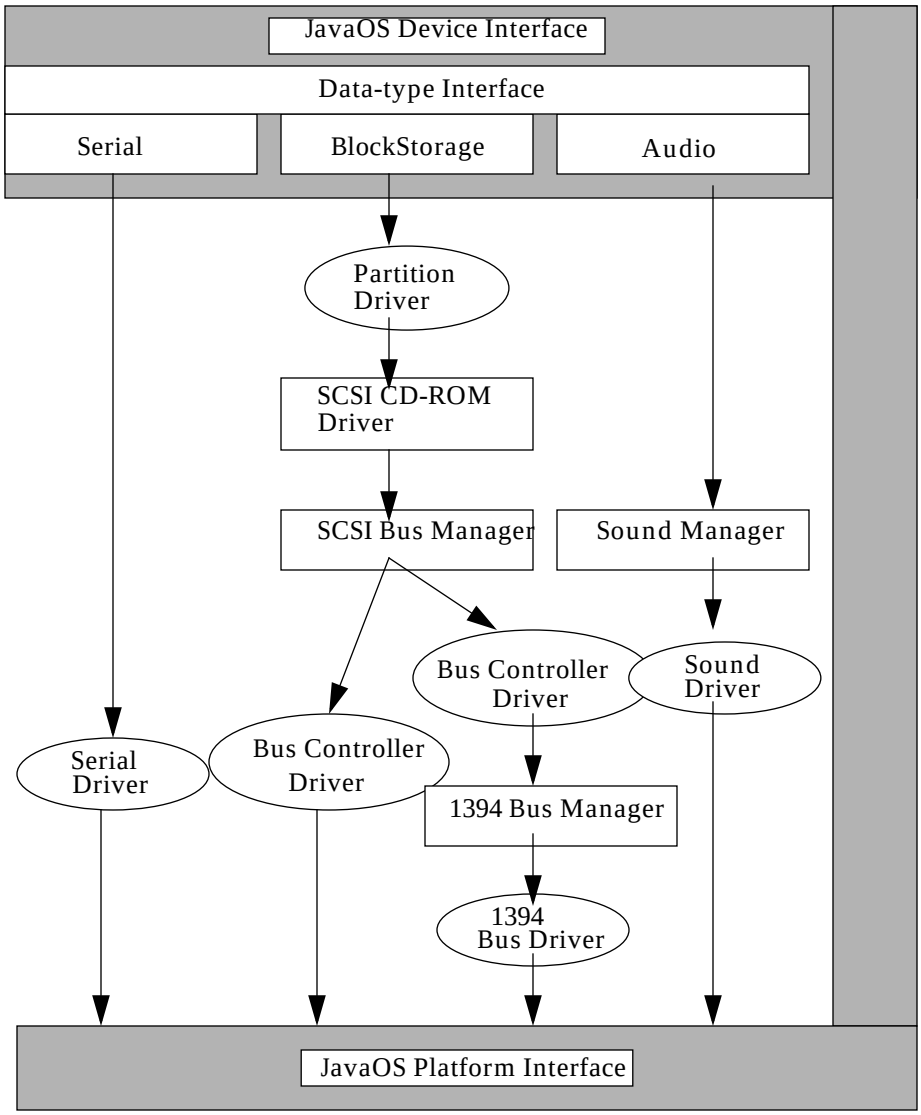


FIGURE 2-13 JDI Layers

JDDK JavaOS for Business Device Interface

The JavaOS Driver Development Kit (JDDK) contains a series of device interfaces, exceptions, handles, and driver classes that are collectively called the Java Device Interface (JDI) for DDK.

Extending the JDI

The JDDK enables driver development for the standard set of JDI devices, but is also extensible. To add JDI support for a device named *widget* for example, the following steps are taken:

- Create the widget *interface*, extending a current device interface. If the widget device represents a totally new category of device, extend the generic Device interface.
- Create the set of unique *exception* classes that pertain to a widget device.
- Create the set of unique *event* classes and interfaces that pertain to a widget device. These classes use the beans design pattern.
- Create the widget device *handle* class named `WidgetDeviceHandle`, extending the handle hierarchy in the same manner as the device interface hierarchy was extended.
- Create the widget device *driver* abstract base class named `WidgetDeviceDriver`, extending the driver hierarchy in the same manner as the device and handle hierarchy was extended.

The following is an example of extending the JDI Framework.

Create the Device Interface(s)

```
public Interface Widget{...}
```

Create the Exception(s)

```
public class Widget xxxException{...}
```

Create the Event(s)

```
public class Widget xxxEvent{...}
```

Create the Event Listeners

```
public class Widget xxxEventListener{...}
```

Create the Handle(s)

```
public class WidgetHandle extends xxxDeviceHandle implements
WidgetInterface{...}
```

Create the Driver(s)

```
public class WidgetDriver implements WidgetInterface{...}
```

The remainder of this section provides an overview of the JDDK JDI standard device interfaces, exceptions, handles, and drivers.

Device Exceptions

The JDI reports error conditions using *exceptions*. Device exceptions are thrown by drivers and handles and contain a reference to the handle in use when the exception occurred.

Device Exception Hierarchy

Figure 2-14 illustrates the DeviceException class hierarchy.

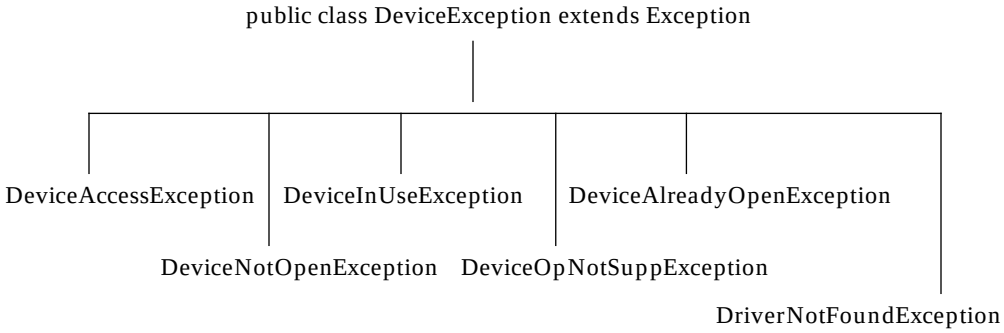


FIGURE 2-14 Device Exception Hierarchy

`DeviceAccessException`

- A `DeviceAccessException` is thrown when an operation is attempted that is not allowed by the current client. An operation is disallowed if the device was not opened with the required action string(s) or if the operation is privileged as determined by platform security management.

`DeviceInUseException`

- A `DeviceInUseException` is thrown during an attempted open when a device is already opened in an exclusive access mode. Clients catching this exception can choose to re-try the open again, hoping that the current device owner issues a close.
  To help initiate the close by the current owner, JDI classes automatically emit an event to the owner requesting the close on behalf of the new client.

`DeviceAlreadyOpenException`

- A `DeviceAlreadyOpenException` is thrown when a second open is attempted on the same open device handle.

`DeviceNotOpenException`

- A `DeviceNotOpenException` is thrown when a close operation is attempted on a handle that is not open.

`DeviceOpNotSuppException`

- A `DeviceOpNotSuppException` is thrown by a driver when an attempted operation is not supported by the device driver.

Device Event Design Pattern

A device event is defined using the Java Beans *event design pattern*. This design pattern enables the handle to be used as a Bean within an Application Builder, for instance.

Following this pattern of interface and object design, each event object and listener is defined as a descendent of the following:

`java.util.EventObject`

`java.util.EventListener`

In addition, each *event object* is dynamically enabled using a notification method defined as follows:

```
public void notifyOnXXXEvent(boolean enable);
```

Each *device handle* stores a private boolean variable per *event kind*. The notify method sets the variable, while the `handleOSEvent` method checks its value before event distribution is invoked.

Devices

A device is defined by its interface. All device interfaces are composed of a series of methods that can be subdivided into groups called *actions*. Each interface in the JDI hierarchy defines its particular actions using an array of strings called actions.

The most basic device action supported by all devices in the JDI hierarchy is the `getInfo` action. The `getInfo` action allows clients to ask a device for the set of device actions, and open handles. The `getInfo` action is defined in the Device Interface:

```
public static final String[] actions = {getInfo};
```

A streaming device for example, supports (in addition to the `getInfo` action) the following set of actions:

```
public static final String[] actions = {read, write};
```

A disk driver must support the format action defined in the disk device interface.

```
public static final String[] actions = {format};
```

Device Interface Hierarchy

Figure 2-15 defines a subset of the device interface hierarchy:

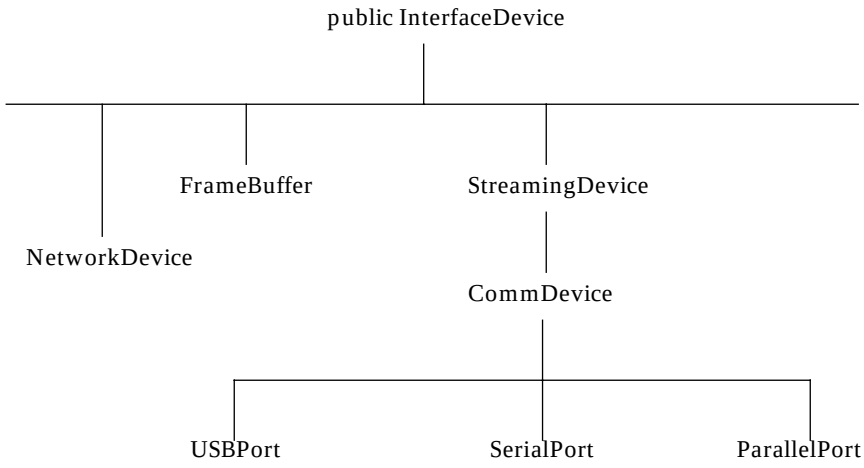


FIGURE 2-15 JDI Device Interface Hierarchy

Each device in the hierarchy defines an interface to be implemented by a driver and a handle.

Device Handle Class

Each device handle implements the complete set of methods and events associated with a specific kind of device. All handles consume device ownership events that alert clients to the availability of a device.

The complete open and close operations are implemented in the base `DeviceHandle` class as is the device ownership event distribution. Note that descendents of the `DeviceHandle` class may override the `handleOSEvent` *method* to distribute events specific to the descendent.

Each concrete handle class must also supply open and close handle methods where descendent specific open and close behavior can be implemented.

All device handles descend from the basic handle definition found in the `DeviceHandle` abstract base class.

A `DeviceHandle` is defined as follows:

```
public abstract class DeviceHandle implements Device, OSEventConsumer
{...}
```

The `DeviceHandle` class contains the following reference information:

- *Service Connection.* A connection via the JSL to the device driver associated with the advertised service. The connection is obtained during handle construction using the JSL `findService` method.
- *Service Advertisement.* An advertised service registered in the JSD interface namespace. Advertisements can be used to obtain a Service Connection object from the JSL.
- *State.* The state of a device handle is either open or closed. The handle also records whether or not the device was opened in exclusive mode. All handles are constructed in the closed state.
- *Device Driver.* The reference to the device driver instance in use by this handle. The device driver reference is obtained during the open process from a Service Connection object.
- *Actions Allowed.* The set of strings naming device actions such as `read` or `write`. Before a handle passes a method invocation on to the driver, the proper device action is checked. If the action required for this method invocation is not found, a `DeviceAccessException` is thrown.
- *Lock.* A reader-writer style lock is used to control the desired amount of handle re-entrance. Handle open and close methods are coded to acquire the write lock before processing the open. Most other device operations such as streaming handle `read` request a `read` operation on the handle lock. Protected methods are made available to sub-classes that can acquire and release the lock for either `read` or `write` operations.
- *Event Listeners.* A vector of beans-style event listeners. Each device handle is suitable to be treated as a Bean. The handle uses the `javax.system.events` package to consume events from the JDI and the device driver. Consumed events are then distributed to individual event listeners. Each kind of event is dynamically enabled and disabled as well. Disabled events are discarded. The `handleOSEvent` method may choose to grab the handle `read` or `write` lock before distributing events to listeners.

Device Handle Class Hierarchy

Figure 2-16 shows the defined subset of the device handle class hierarchy.

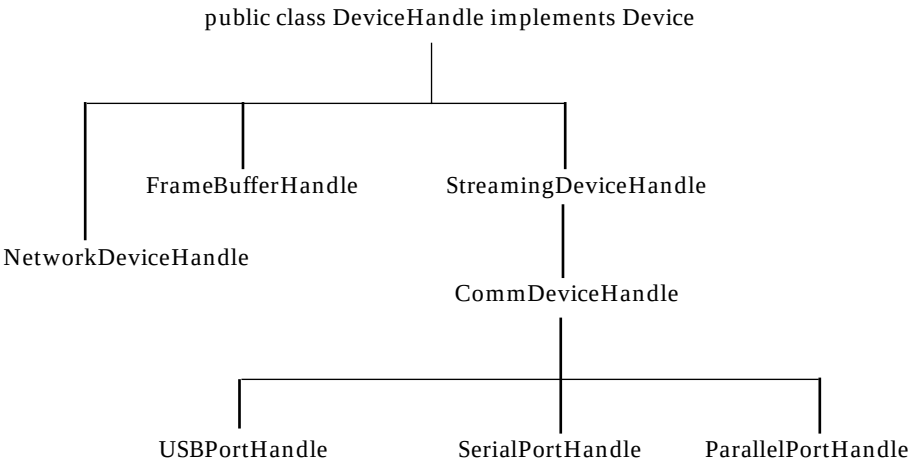


FIGURE 2-16 JDI Device Handle Class Hierarchy

Device Driver Class

The base device driver provides the following functionality:

- *Stores Vector of Open Handles.* The `DeviceDriver` class implements the `getOpenDeviceHandles` method by returning a copy of each handle reference.
- *Remembers Exclusive Open Handle.* If the driver was opened for exclusive use, the exclusive handle (i.e. device owner) is remembered and returned to clients using the `getCurrentOwner` method.
- *Returns Device Action Definitions.* The `DeviceDriver` class provides a default implementation of the `getDeviceActions` method by returning the `xxxDevice.actions` array of strings.
- *Produces Device Ownership Events.* At the appropriate point during open and close processing, the `DeviceDriver` base class produces various device ownership events that alerts clients as to the availability of a device. Concrete implementations of abstract driver classes are given the choice as to the synchronous or asynchronous flavor of event producer.

Device Driver Class Hierarchy

Figure 2-17 defines the subset of the device driver class hierarchy.

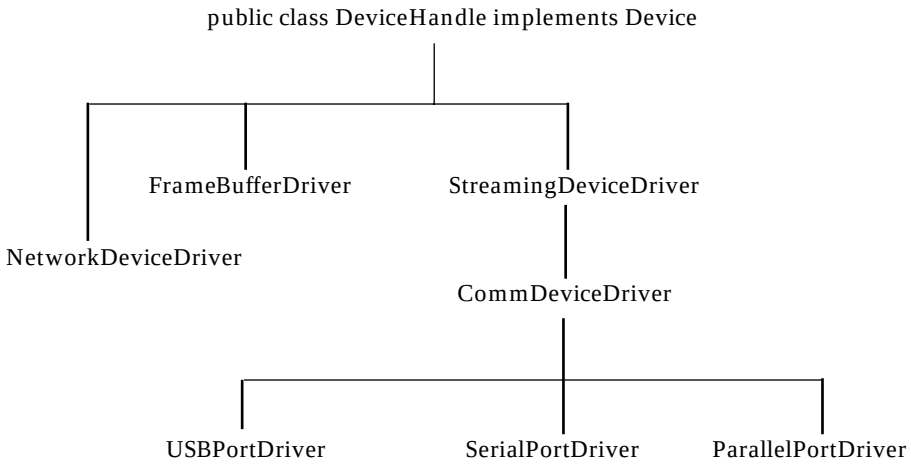


FIGURE 2-17 JDI Device Driver Class Hierarchy

Each driver in the hierarchy implements the corresponding device interface in the device interface hierarchy. The handle, interface, and driver hierarchies mirror each other.

Accessing a Device Handle

This section presents an example of how a device is accessed using a JDI object known as a *device handle*. A device handle is a Java bean that connects a device client (such as an application) to a device driver. Once a handle is connected to a device driver, a client can open the device referenced by the driver, and then proceed to use the capabilities of that device.

The example presented here depicts a handle usage scenario with a fax device and a client application.

Finding a Device

All devices must first be discovered and then published (in the JSD *device* namespace) before clients (such as applications) can find and use them. The act of publishing a device kick-starts the subsequent JSL process of matching a driver to a device. Once the JSL has matched a driver to the device, the matching driver public interfaces are automatically *advertised* in the JSD *interface* namespace.

An interface advertisement references the driver, and the driver references the device. Finding a device is the process of finding a device driver *public interface advertisement*. The following figure shows the relationship between drivers, devices, and interfaces.

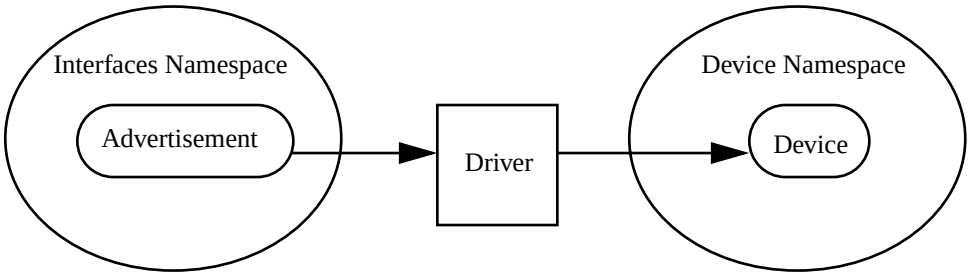


FIGURE 2-18 Drivers, Devices, and Interfaces

Finding a Fax Advertisement

Interface advertisement objects are stored in a special property (named Service.ADVERT_PROP_NAME) of entries in the interfaces namespace. Advertisement entries are arranged in a hierarchical fashion using a scheme based upon Java™ package and interface names. For example, an interface to a fax device is defined as follows:

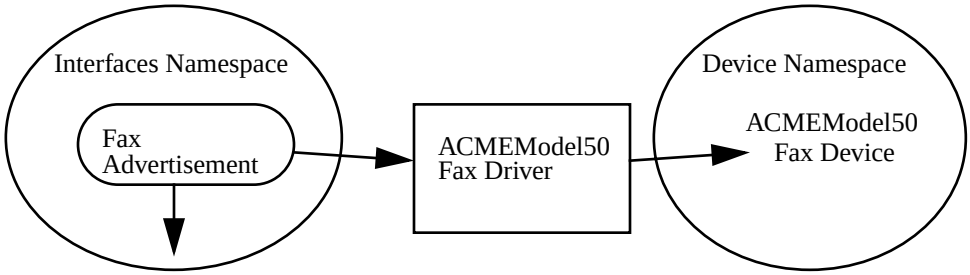
```
package javaos.javax.system.jdi;

public interface Fax extends StreamingDevice {
}
```

A combination of the package name `javaos.javax.system.jdi` and interface name fax is used to create an interface entry with the pathname: `/interface/javaos/javax/system/jdi/fax`. Each fax discovered and matched to a driver, results in an fax advertisement entry being created as a *child* of the fax interface entry.

The driver supplies device names in its business card, such as `ACMEModel50Fax`, which in this case would cause the entry `/interface/javaos/javax/system/jdi/fax/ACMEModel50Fax` to be created.

Figure 2-19 shows the relationship between fax drivers, fax devices, and fax interfaces.



/interface/javaos/javax/system/jdi/fax/ACMEModel50Fax

ServiceAdvertisement a = `advertisementEntry.getPropertyValue Service.ADVERT_PROP_NAME);`

FIGURE 2-19 Fax Drivers, Fax Devices, and Fax Interfaces

Techniques for Finding a Fax Address

An application can find a fax by using one of the following three techniques:

- JSL Enumeration
- JSD Query
- JSD Event

JSL Enumeration

The JavaOS Service Loader (JSL) provides a public static method to return the set of advertisements associated with an interface name. Given an interface name, applications can use the following static method defined in the ServiceLoader class to obtain an *enumerated* list of advertisements:

```
public static Enumeration getAdvertisements(String interface) throws
ServiceException;
```

For example, to obtain an enumerated list of fax advertisements, use the following code sample:

```
ServiceAdvertisement a;
String interfaceName = "/javaos/javax/system/jdi/fax/";
Enumeration faxes;

faxes = ServiceLoader.findAdvertisements(interfaceName);
while(faxes.hasMoreElements()) {
a = (ServiceAdvertisement)faxes.nextElement();
}
```

JSD Query

An application can also use the JSD Query mechanism to cycle through a set of advertisement entries. Applications using this method to find advertisements must read the property service `ADVERT_PROP_NAME` attached to each advertisement entry to actually obtain a ServiceAdvertisement object. The following code sample accomplishes this:

```
import javaos.javax.system.database.Query;
import javaos.javax.system.database.Tree;

ServiceAdvertisement a;
String interfaceName = "/javaos/javax/system/jdi/fax/";
Tree t = SystemDatabase.getSystemDatabase();
Entry advertisementEntry;
Query q = new Query(t.findEntry("/interface" + interfaceName),
Query.CHILDREN);
while ((advertisementEntry = q.nextMatch()) != null) {
a=advertisementEntry.getPropertyValue(Service.ADVERT_PROP_NAME);
}
```

JSD Events

The third method for finding advertised interfaces relies on the JSD event production capability. Each time an advertisement entry is added to the interfaces namespace, an application can receive an event that contains a reference to the advertisement entry. In order to receive events from the JSD however, an application must first subscribe to events that pertain to a particular sub-tree of the database.

For example, the filter string containing the value: `/interface/javaos/javax/system/jdi/fax/` matches events that pertain to fax advertisements only.

The following code sequence subscribes an application to fax advertisement entry insertion events:

```
import javaos.javax.system.events.*;

public myFaxInsertionEventConsumer implements OSEventConsumer {
public myFaxInsertionEventConsumer() {
int producerCount = OSEventProducer.subscribeConsumer(this,
OSEventProducer.sharedConsumer);
}

public String getName() { return “sampleFaxConsumer”; }
public Class[] getEventKinds() {
Class[] classesIWant =
OSEventProducer.buildEventKindArray(“EntryInsertEvent”);
return classesIWant;
}
public boolean wantsInstanceOf(Class eventKind) {
return true;
}

public String getFilter() {
String interfaceName = “/javaos/javax/system/jdi/fax/“;
return “/interface” + interfaceName;
}

public int handleOSEvent(OSEvent insertionEvent) {
ServiceAdvertisement a;
EntryEvent e = (EntryEvent)insertionEvent;
Entry advertisementEntry = e.getEntry();
a=advertisementEntry.getPropertyValue(Service.ADVERT_PROP_NAME);
System.out.println(“Found an advertised fax!”);
}
}
}
```

Device Handle Construction

Given an advertisement to a device such as a fax, an application can then construct a device handle that matches the device type. If the handle type and the device type do not match, an exception is thrown. In the case of a fax, a fax handle should be constructed. The following code sequence constructs a fax handle given a fax advertisement:

```
import javaos.javax.system.jdi.FaxHandle;
FaxHandle h;
ServiceAdvertisement aFax = getTheFax();
h = new FaxHandle(aFax);
```

Every device handle contains a set of device actions. An action is a named set of device methods (or capabilities). When constructing a handle, an application can choose to override the default set of actions by passing a specific set to the constructor. Action names are defined as public constants in a device public interface. For example, the Fax interface definition contains the following public constants:

```
public static final String[] actions = {“init”, “send”, “receive”};
```

The following code sequence constructs a fax handle that allows an application to only *receive* faxes:

```
import javaos.javax.system.jdi.FaxHandle;
FaxHandle h;
String[] faxActionsToAllow = new String[1];
faxActionsToAllow[0] = Fax.actions[2];
ServiceAdvertisement aFax = getTheFax();
h = new FaxHandle(faxActionsToAllow,aFax);
```

If the application attempted to send a fax using a receive handle, an exception is thrown.

Opening the Device

The next step required of an application using a device is to *open* it. All handles support an open method declared in the `DeviceHandle` interface. The following two open methods are available:

```
public final void open() throws DeviceException;
public final void openExclusive() throws DeviceException;
```

The first open method (open) is invoked to open the device in a *shared* mode. That is, the application (and the device) can tolerate more than one open handle (client) at a time.

The second open method (openExclusive) is invoked when only a single open handle is desired or supported.

Opening a device may cause the code of the device driver to be loaded into memory. The JSL only loads the driver if necessary.

Using an Open Device

Using an open device, means invoking device handle methods, such as `send` and `receive`. Each time an application invokes a capability method of a device handle, the handle checks its action list. If the desired action is not supported, the handle throws an exception.

If the action is supported, the handles calls the very same method in the driver. The following figure illustrates the process:

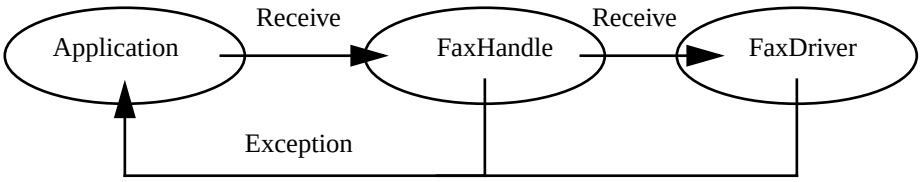


FIGURE 2-20 Using a Device Handle to Receive a Fax

Exceptions thrown by the driver during the process of receiving a fax are also propagated back to the application.

Closing the Device

When an application no longer needs access to a device, the device handle should be closed. Closing a handle may cause the JSL to unload the driver if no open handles remain. The following method is defined in the DeviceHandle class and is supported by all types of handles:

```
public final void close() throws DeviceException;
```

3

JavaOS Platform Interface (JPI)

Introduction

This chapter describes the JavaOS Platform Interface (JDI) for JavaOS for Business software including the:

- The Role of JPI
- The JavaOS for Business Memory Model
- JVM System Interface Summary
- Interrupt Management
- The Boot System
- The Microkernel
- The Runtime
- JavaOS for Business Memory and DMA Classes
- Interfaces
- JDI and JPI Interfaces
- Microkernel Interfaces

JavaOS for Business is optimized to run Java code on a variety of computing and consumer platforms. Rather than building on top of an existing operating system, the software provides a Runtime specifically tuned to run Java applications.

The software is built from a combination of native code (instruction set and hardware platform specific) and Java byte-code, which is platform independent.

The software defines a platform as a CPU, a physical memory map, and any permanently attached devices, buses, and slots. An example of a permanent device is a PCMCIA bus and slot. A card in the PCMCIA slot is considered a transitory device, and falls outside the definition of platform.

The platform-independent component of the system is called the Runtime, and the platform-dependent portion is called the Microkernel.

Runtime requirements are a superset of the Java Virtual Machine (JVM) Runtime requirements. This chapter details both the JVM system interface and the interface extensions necessary to support JavaOS for Business Runtime. It also provides a high-level overview of the memory and interrupt architecture, a well as the Microkernel and Boot layers.

Figure 3-1 illustrates the layered architecture:

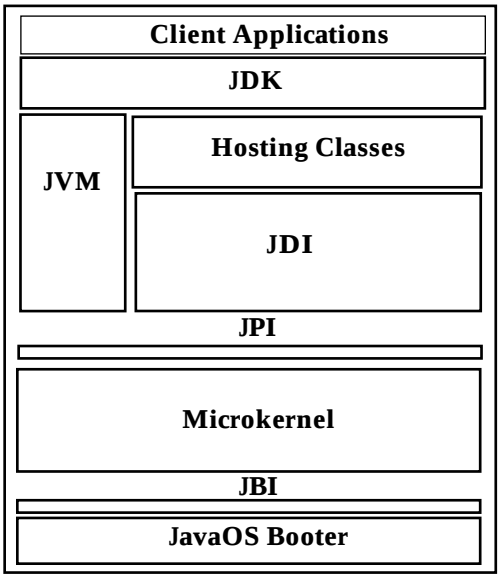


FIGURE 3-1 The JavaOS for Business Layered Architecture

The Boot Layer

The first software layer contains the booting system. Boot systems for network, ROM, RAM, CD-ROM, floppy, and hard disks are all possible. Boot systems for network computers however, may differ widely from embedded device boot systems.

Regardless of these differences, all boot operations are normalized to a standard interface called the JavaOS for Business Boot Interface (JBI). This interface is used both by the Microkernel and the Runtime layers of the software.

The JBI defines a bi-directional interface as well as an OS execution environment. The bi-directional interface provides a function that passes control to the software and also allows the software to request services of the booting system. The following illustration depicts the relationship between the Microkernel, the Boot system, and the JBI interface:

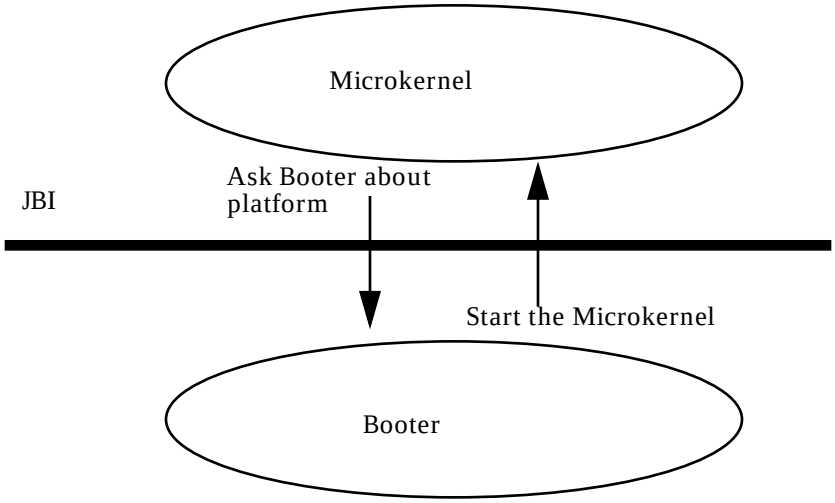


FIGURE 3-2 How the JBI Determines Platform Data

The JBI is designed to give product designers flexibility when making booting decisions. Some JavaOS for Business booting implementations may leverage an existing PROM standard such as OpenBoot or a PC-BIOS. Some embedded products such as a smart phone may require a complete custom PROM.

Please refer to Chapter 7 for a complete discussion of the JBI.

The Microkernel Layer

The second layer in the architecture contains the Microkernel. The Microkernel supports the virtual machine's system functions, a comprehensive debugging API, and the Runtime native methods. The required functions fall into the following categories:

- Threads
- Monitors
- File System

- Exceptions
- Timing
- Native Code Library Management
- Memory Management
- Interrupts
- DMA
- Debugging
- Miscellaneous Platform Control

Threads, monitors, files, timing, libraries, and exception APIs are grouped together and collectively called the Java Virtual Machine (JVM). These APIs are designed to support the JVM.

A debugging API exists that supports both native and java code. This debug API enables a remote host to debug a JavaOS for Business target over a variety of transports, such as ethernet, parallel, and serial.

The protocol that operates over the transport is independent of the host platform and able to accommodate many tool vendors. Two levels of debugging are supported: system and thread.

A system-level operation treats the target as a whole system. When a thread hits a system breakpoint, the entire target is stopped. A thread-level operation, however, only operates on a thread; that is, the operation would only stop the thread that hit the breakpoint.

The memory, interrupt, DMA, and miscellaneous remaining APIs are designed to support the device driver model and are called the Runtime Native Methods.

These first two layers (booting and Microkernel) are considered platform dependent. The code in these layers is composed of a combination of C and assembly, some of which is CPU-dependent.

The Runtime Layer

The layers above the Microkernel begin the platform independent portion of the operating system. These layers are collectively called the JavaOS for Business Runtime. The Runtime includes the JVM as well as code to support AWT, networking, and file-related IO classes. The Runtime also includes support for device drivers written in Java. These drivers are an important client of the JPI. See Chapter 2, The JavaOS for Business Device Interface (JDI), for a more detailed discussion of Java-based device drivers.

In the software, AWT is layered upon a window system occupying space in the Runtime layer, as shown in the figure below:

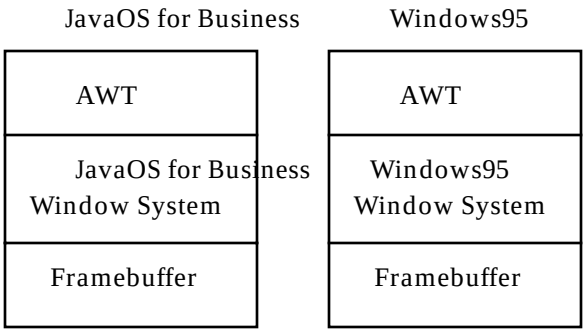


FIGURE 3-3 Layering in the JavaOS for Business software

JavaOS for Business Runtime

The Runtime is designed to run on hardware-limited platforms. For example, the Runtime does not require a Memory Management Unit (MMU) to map virtual addresses to physical memory addresses, nor does it require memory protection.

The Runtime lets developers determine whether the underlying Microkernel should use an MMU or enforce memory protection. For example, the Microkernel uses an MMU to make several noncontiguous memory regions appear contiguous to that version of the Runtime.

The features implemented in the software by the underlying Microkernel will most likely be shaped by the product under construction. For instance, an implementation destined for a phone will need a small, simple Microkernel. However, a network computer may require a more complex Microkernel implementation.

The Role of JPI

The JPI is an interface designed to provide flexibility between the Microkernel and the Runtime layers. It allows separate teams and/or companies to work on the Microkernel and Runtime components of the OS in parallel. From a product point of view, one Microkernel implementation could be tuned for hard realtime environments, while another Microkernel could be designed for a network computer. Figure 3-4 shows the relationship between the Runtime and the Microkernel:

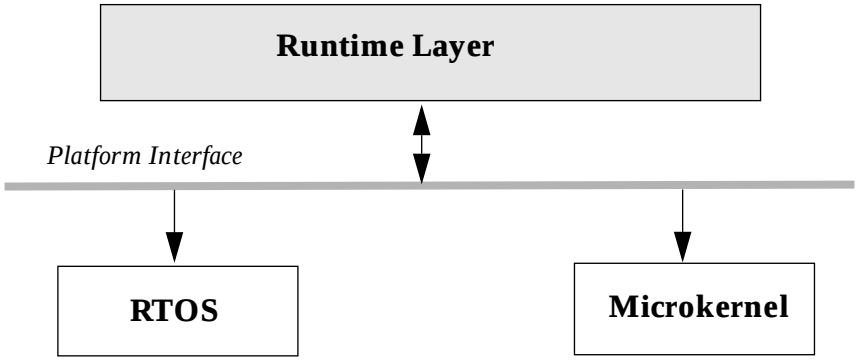


FIGURE 3-4 Runtime and Microkernel Relationship

The Microkernel was designed with a network computer in mind. It has implemented the JVM system functions and provided just enough device-related native methods to support linked drivers.

The JDI supports a formal, portable device driver model. See chapter 2 for more information.

Note: Loadable device drivers are supported by the JavaOS for Business System Loader (JSL). See Chapter 6, JSL for more information

The Java API

The highest level interface—the Java Development Kit (JDK)—defines the Java API. This API supports applications such as browsers and browser applets, and is the same for all Java Runtimes including JavaOS for Business, and UNIX® implementations.

The system architecture is composed of a number of Java and native API's. Some of the API's are optional while others are required. Implementation teams must implement all of the required API's and whole subsets of the optional ones. Whole Java API subsets must include complete classes, not just a few methods in a class. Native interfaces must be completely implemented with no short cuts. An implementation that uses half of the hash table class and part of the windowing APIs, for instance, is not considered a sanctioned use of the software.

JPI Clients

JPI clients are divided into two groups: native and Java. The primary native client is the JVM. The JVM views the platform interface as a set of C-based routines providing the raw platform services required by the virtual machine. This collection of services is called the *JVM System Functions.*

The Java-based clients include Java-based device drivers, and various system software components such as the Expansion Bus Manager. A Java-based client views the platform interface as a set of support classes that are implemented as a combination of Java and native methods.

The JPI composition is illustrated in Figure 3-5.

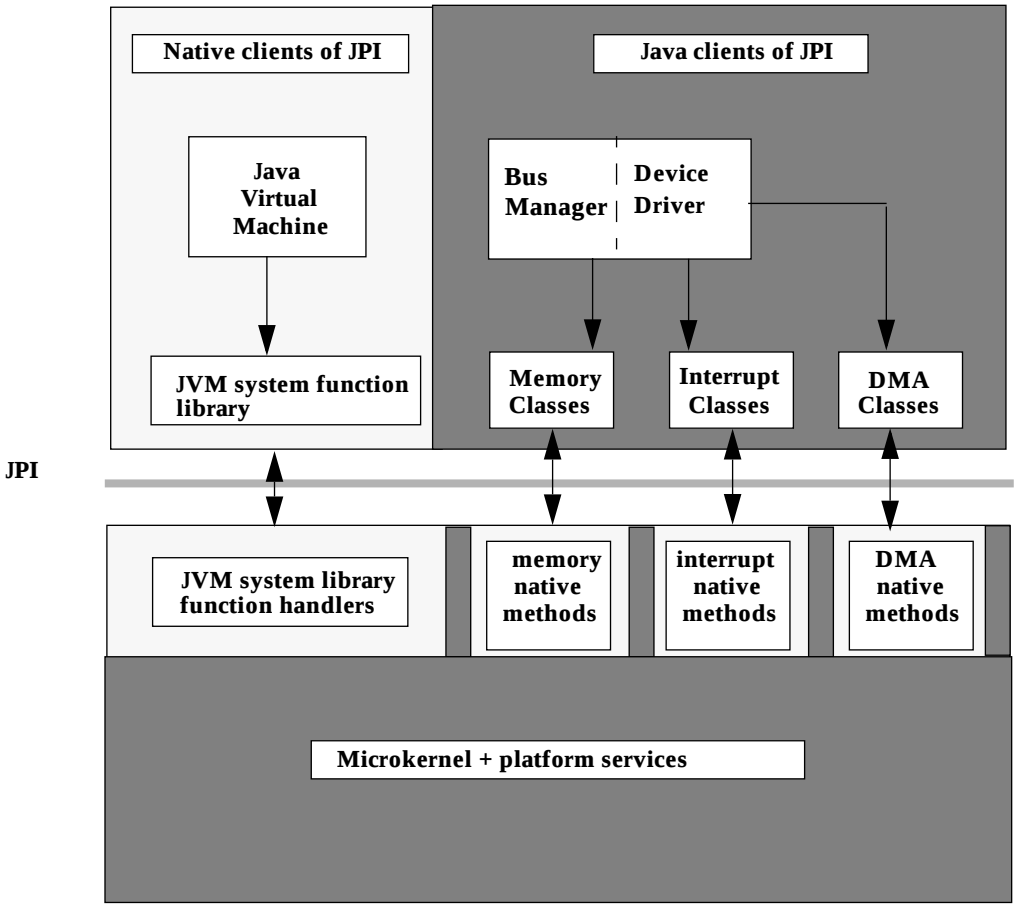


FIGURE 3-5 Clients of the JavaOS Platform Interface

From the Microkernel point of view, the Microkernel provides native, C-based handlers that execute on behalf of the native and Java clients. The C-based handlers are platform-dependent. The client-invocation side of the interface is platform independent.

JPI and the JavaOS for Business Device Interface

Since all device drivers managing access to a physical hardware device use the services of the JPI, the JPI insulates both the Runtime and device drivers from platform dependencies. However, platform independence doesn't mean bus-independence. Each device driver operates in the context of an expansion or I/O bus.

Platform Tuned Device Drivers

While the JPI insulates drivers from platform dependencies, it does not prevent a driver from garnering specific information that can help it optimize performance. Each component of the JPI publishes methods that return platform-specific information. The driver developer can use this information to optimize driver performance.

The Bus Managers and the Java System Database

Bus managers use information stored in the persistent server part of the JSD to retrieve parameters describing how to communicate with any device. The information is actually determined by the bus manager when it connects with the device, or when it needs to establish a connection, and is then stored both in the persistent version of the database that exists on the server and on the transient database version that exists on the client.

Details about what kind of information is stored and how it is used is explained in more detail in Chapter 4, The JavaOS for Business JSD.

The JavaOS for Business Memory Model

All memory models build upon the notion of addressing. Addressing is the process of identifying a memory location. The most fundamental unit of addressing in the software is the physical address. A physical address identifies a unique memory location. A physical address is not translated by the MMU; rather it is a raw memory location, identifying what may be ROM, RAM, or I/O memory.

An address bus initiates the movement of code and/or data from one platform device to another using physical addresses. Physical addresses are generated by devices acting as bus masters on the set of address buses connected to the platform. Bus masters are those devices capable of generating addresses. Pure slave devices

can respond to an address but not generate an address. Some devices can both generate and respond to addresses. Code and/or data is moved on a data bus in response to a bus master generating addresses on an address bus.

The software views the CPU(s) connected to the platform as just another device generating and possibly responding to physical addresses. A memory-mapped bus extends the CPU's view of physical memory to include the devices attached to that bus. Attached devices can be accessed by general purpose load and store instructions and are called memory-mapped devices. Memory-mapped buses include PCI, Sbus, and VMEbus. Not all buses and devices fall into this category, however.

Some buses require special instructions to generate an address. The x86 architecture supports the notion of port addressing. A port is a form of physical address that can only be generated using special `in` and `out` CPU instructions. SCSI is an example of an I/O bus that is not memory-mapped and responds to special addresses that identify the bus, device, and logical unit within the device.

The design of the software memory model is based upon the following requirements:

- *Java and JavaOS for Business-centric solution*. The software memory model is exclusively designed for Java-based software and for the Java operating system. It is not designed to be OS and programming language independent. Its sole purpose is to provide JavaOS for Business-based software secure and flexible memory access.
- *Microkernel independent*. The software supports the notion of replaceable Microkernel's tuned for a computing platform. The software memory model is constructible upon industry standard Microkernels such as those available from Real-Time Operating System (RTOS) vendors.
- *CPU and MMU independent*. The software memory model does not depend upon any feature provided by a CPU or memory management model. The model is constructed and supported completely in software. The software makes no use of the supervisor/user mode features of a CPU, for example, because it would unnecessarily limit the number and kinds of platforms capable of supporting JavaOS for Business.
- *Access to memory is limited to bounded regions as determined by the software.* The system gives device drivers memory access, but not the ability to randomly touch any region of memory. The software memory model adheres to the original spirit of Java memory management by encapsulating memory regions in objects.
- *Support Virtual Memory*. The software memory management allows for the possible existence of a paged memory system supported by the Microkernel. The software memory model insulates device drivers from the CPU and MMU specific nature of paged memory management.
- *Portable Addressing Scheme.* The software memory management supports the notion of portable memory addressing. The addressing scheme is not be associated with any one CPU or MMU. The software addressing scheme insulates programmers from the CPU's representation of an address. For example, 64-bit,

36-bit, or 32-bit CPU pointers are hidden from the Java software, providing a new level of memory access abstraction. The software memory model also normalizes data ordering memory issues and other problems that typically reduce software portability.

- *Extensible Addressing Scheme.* Memory management supports the notion of extensible memory addressing. The addressing scheme accommodates the addition of I/O and expansion bus addresses. The size and representation of I/O and Expansion bus addresses is abstracted by the JOS software memory model.

The C programming language lets a programmer directly address memory using pointers and pointer arithmetic. Pointers allow access to critical areas of memory such as I/O space.

Most modern operating systems attempt to limit such I/O memory access to device drivers. The limitation is enforced using the CPU's memory management unit. Device drivers are typically executed in the processor's supervisor mode, and the OS programs the MMU to limit I/O memory access to supervisor mode code only. This enforcement prevents user-mode applications from unsecured I/O memory access, but nonetheless allows a device driver to easily crash an operating system.

The Java programming language eliminates the direct manipulation of memory by encapsulating all access into objects, classes, and automatic free memory collection. Eliminating pointers makes Java a more robust programming language than C and reduces the number of memory-related bugs.

The memory model and services of the JPI have perhaps the biggest impact on portable system software such as device drivers. The software provides a controlled method of memory access that adheres to the Java philosophy of robustness but provides enough flexibility to support device drivers, DMA, and direct access frame buffers.

Note: The software memory model is *not* available as part of the Java programming language. Further, the software makes no additions to the language or core classes in order to implement a portable memory model.

Physical and Virtual Address Space

A computing platform's physical address space is the range of physical addresses available on all the connected buses. The processes and rules governing physical addressing and access are known as physical memory management.

In addition to physical addresses, memory, and memory management, most CPUs with the help of a address translation device MMU, (Memory Management Unit) also support virtual addressing and virtual memory.

A physical address is not translated by the MMU; rather it is a raw memory location, identifying what may be ROM, RAM, or I/O memory.

A virtual address is an abstract identifier of a memory location that must be translated into a physical address before the access occurs. A range of virtual addresses is called virtual memory. The total possible range is called the virtual address space. A fundamental addressing rule of by the software is that all Runtime software including the JVM itself resides in a single virtual address space.

The software provides both physical and virtual memory management programming interfaces. Figure 3-6 illustrates the relationship between the physical and virtual address spaces:

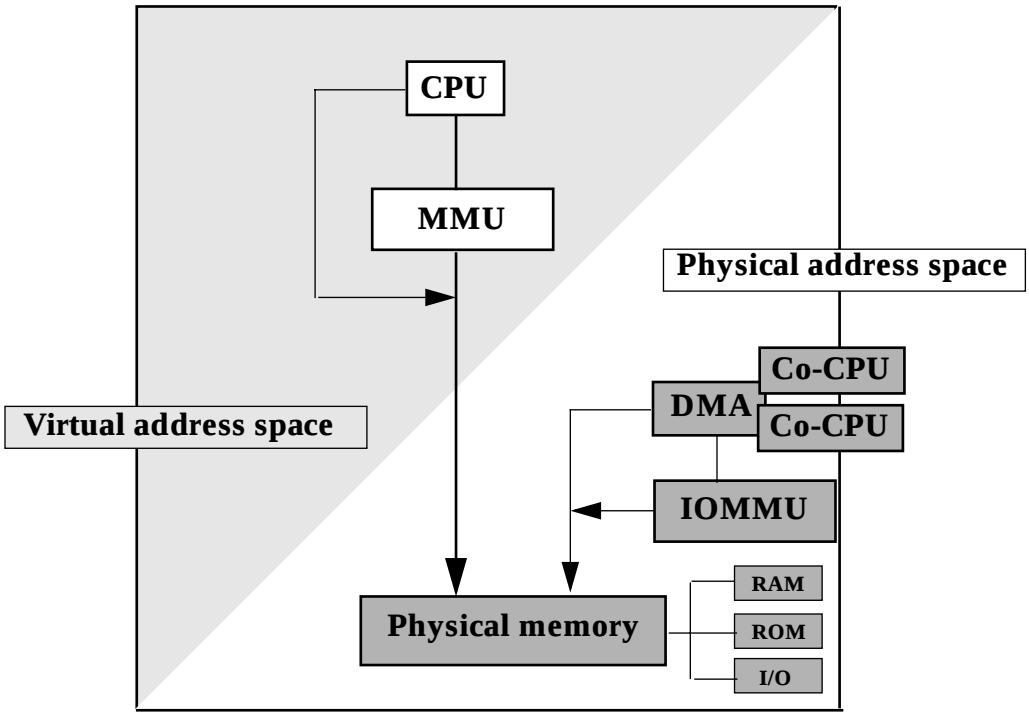


FIGURE 3-6 JavaOS for Business Address Spaces

The CPU and all software accessing memory through the CPU use virtual addresses. Even CPUs without MMUs are assumed to be using virtual addressing. At some point during the CPU's memory access, the virtual address is translated into a physical address that uniquely identifies a location in the physical address space.

Sometimes the translation from virtual to physical involves an MMU. Some JavaOS for Business software implementations, however, do not use an MMU, yielding a one-to-one relationship between a virtual and physical address.

From the software memory management point of view, when a secondary bus-master accesses memory, the address is always considered a physical address—even when an IOMMU is available to map memory.

When an IOMMU is programmed to map a CPU virtual address, a temporary one-to-one relationship between the virtual and the physical is established. For a disk device the term of the relationship may be just a single DMA transfer. In the case of an ethernet device, the IOMMU may be programmed to map a ring buffer on behalf of the DMA controller for the duration of the driver's installment. The use of a CPU or IOMMU to map virtual to physical addresses is considered a platform dependent decision under the control of the native Microkernel.

The number and kinds of devices that may be attached to a platform is nearly limitless. Device intelligence can range from the simple to the very complex.

Some devices can only generate addresses aligned to designated boundaries. Some devices (like the CPU) may cache memory accesses. Caching can improve overall CPU performance rather dramatically, but presents still more memory management problems.

Not all devices have access to all of physical memory. For example, a DMA controller may be limited to 16-bits of physical addressing, or a legacy PC device may only address the lower megabyte of memory.

Address Space Management

The JavaOS for Business software provides a set of tools to manage memory addressing and promote portable software. These include tools and fundamental management APIs from which drivers can manage a device in a portable fashion.

The most fundamental set of tools is a collection of addressing classes. These classes are of interest to both driver developers as well as OEMs porting the OS to a new CPU and/or Microkernel foundation. The address classes define the fundamental size and representation of any address used in the operating system, providing a set of extensible address types that may be used to define more abstract kinds of addresses.

DMA memory, physical memory, and virtual memory are all addressed by addresses from the same *Address* abstract class. Since there are significant examples of systems that have addresses with a non-power-of-two number of bits, the *Address* class supports addresses with any number of bits between 1 and 64.

Unsigned Values Representing Addresses

Internally to the class, an `Address` object is represented as an unsigned value, along with an indication of the number of bits. Since Java does not have unsigned arithmetic as part of its foundation classes, it was necessary to develop some simple efficient algorithms for addition and comparison of addresses.

The `Address` abstract class is abstract mainly because it is most efficient to use different algorithms for unsigned arithmetic, depending upon the number of bits used to represent the address value. There are two sub-classes, `Address32` for addresses up to 32 bits, and `Address64`, for addresses up to 64-bits.

Physical Space Address Layout

The physical address space for the system is the domain of physical addresses available on the computing platform. The system recognizes a physical address space divided into the following contiguous memory regions:

- *ROM*. Read Only Memory is an area of contiguous physical memory that may or may not hold the software. The ROM may in fact hold data such as built-in fonts. In general the software makes no assumptions about the contents of the ROM. Unlike the case with RAM, the software assumes no bus errors when accessing any portion of the ROM. Please note that the system software views flash ROM as a device that *maps* a ROM physical memory area. The flash ROM driver knows how to direct the flash devices to fill ROM with new values.
- *RAM*. Random Access Memory is an area of physical address space. The system does not assume that all of this RAM area is actually backed by memory devices. In other words, the software is prepared to take an address/bus error in a portion of this area. On the other hand, the software does require some RAM (a half megabyte or more is preferred for basic functionality). The RAM area may in fact be subdivided with each subdivision discontiguous in nature. The RAM area is described as one or more banks of address space. See the JBI chapter for a discussion on physical and virtual memory tables.
- *Fixed I/O*. This area of address space contains any permanently attached I/O devices. This area holds devices that do not negotiate for address space; rather, they assume a hard-coded range of addresses in this area.
- *Expansion I/O*. This area of address space contains devices that are dynamically discovered and that do negotiate for a range of addresses. PCI is an example of a bus whose devices occupy this area of physical address space.
- *Platform Specific Control*. This area of address space contains platform specific control functionality such as power management or physical address space mapping control bits (i.e. some platforms may support the dynamic re-mapping of ROM after the system begins its startup sequence).

Figure 3-7 illustrates a sample 32-bit Physical Address Space:

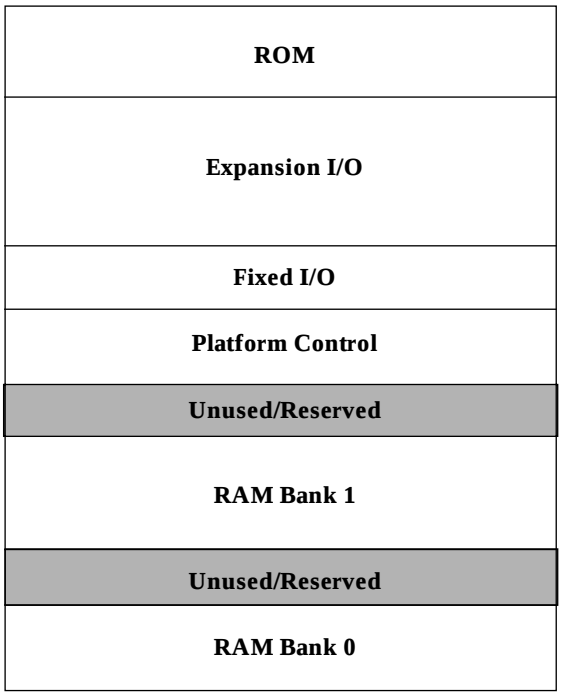


FIGURE 3-7 Sample 32-bit Physical Address Space

The system software makes no assumptions about the placement of areas within the physical address space.

The booting system discovers and presents (via the JBI) the physical address space to the Microkernel upon startup. See Chapter 7, JBI for a complete discussion of the booting sequence.

Virtual Space Layout

The JavaOS for Business software virtual address space is created by the Microkernel layer. Note that the term virtual doesn't necessarily imply paging. Instead, it means that this address space can be used by all software and does not necessarily resemble the physical address space. The software does not assume a one to one correspondence between the physical and virtual address spaces. The use of an MMU to enhance the support of the virtual address space is not assumed either. Rather, MMU usage and page management are seen as Microkernel implementation issues.

The system software operates in a single virtual address space. The physical address space only comes into play when a bus master other than the CPU requires an address. (i.e., a DMA controller or coprocessor). When a physical address is required, the system software asks the Microkernel layer to translate a virtual address range into one or more regions of physical memory.

When addressing virtual memory or physical memory, the Runtime assumes the following:

- Virtual memory accesses may cause a page fault.
- The Virtual address space may be larger than the related physical RAM area.
- Virtual memory management is the job of the underlying Microkernel. The Runtime allocates and uses virtual memory via the JPI. The Runtime manages heaps and the Microkernel manages virtual memory areas as groups of pages. If pages are allowed to fault by the Microkernel, the resolution of the fault is the job of the native Microkernel layer and doesn't involve the Runtime.
- Physical memory accesses never cause a page fault, but may cause a bus error and/or address error.

The virtual address space provided by the Microkernel is subdivided into the following contiguous memory areas:

- *MicroKernel Text Area.* This is an area of contiguous virtual memory containing the text (native code) of the Microkernel layer. This area is assumed to be read-only. The Microkernel may use an MMU to enforce the read-only accessibility if so desired.
- *MicroKernel Data Heap.* This area of contiguous virtual memory holds the dynamic data allocated by native code. This area is read-write. Memory from this heap is strictly for use by the native code portions of the Runtime and by the native Microkernel. This area of memory is not under the control of the Garbage Collector.
- *MicroKernel Stack Area.* This area of contiguous virtual memory holds dynamically created thread stacks for native and/or Java code. The stacks are assumed to be allocated in page size chunks of a Microkernel and processor dependent size.
- *Java Heap Area.* This area of contiguous virtual memory holds the dynamically stored objects. It is patrolled for garbage in both a synchronous and asynchronous fashion by the Garbage Collector. This area grows and shrinks as needed from the MicroKernel Data Heap.
- *Java Method Area.* This area of contiguous virtual memory holds the dynamically stored Java byte code. It is also patrolled for garbage in both a synchronous and asynchronous fashion by the Garbage Collector, and may in fact be a part of the Java Heap Area.
- *Java Constant Pool Area.* This area of contiguous virtual memory holds the per-class or per-interface Runtime representation of the constant pool table in a Java class file.

All the Runtime memory areas are created, managed, and ultimately destroyed by the Runtime. All Java Microkernel memory areas are constructed using memory allocated from the Microkernel data heap. All Microkernel memory areas are created, managed, and destroyed by the Microkernel.

This clear separation of memory areas allows each layer to tune allocation and collection to the needs of its clients and to the environmental aspects of the target product. For example, an network computer computer with a local hard disk may be a suitable platform for paging. The Microkernel can choose to grow stacks dynamically by taking page faults as the stack grows. Pages are dynamically assigned and retired by the Microkernel without knowledge or help from the Runtime. The Runtime is fully insulated from such activity, yielding increased portability.

JVM System Interface Summary

The JVM has a set of functions which perform system service functions. These are broadly grouped as follows:

- Threads functions
- Monitor functions
- File functions
- Runtime functions

These functions are private API's that may be accessed only by OEM's with access to the source code.

Memory Object Construction

The system software encapsulates device- and platform-specific properties in a handful of bus managers which construct memory objects for the vast array of device drivers to use to access memory. The Memory classes let a bus manager construct a generic memory object, which ultimately provides access to one of three fundamental types of memory (as specified by the bus manager):

- Regular memory, which is accessed via normal load and store operations from the CPU. This memory could be directly addressed by the CPU (physical memory), or could be indirectly accessed through an MMU (virtual memory).
- I/O memory, which is accessed via normal load and store operations from the CPU, but which may also require other operations to be done to keep the I/O memory coherent with main memory. Again, this memory could be directly addressed by the CPU (physical memory) or indirectly through an MMU (virtual memory).

- I/O port memory, which is accessed via special I/O instructions (e.g., `in` and `out` instructions on Intel x86-based platforms).

Byte Ordering

To achieve the goal of platform independent device drivers, the JPI must address the issue of CPU-dependent and bus-dependent byte ordering, and big-endian and little-endian characteristics. (Big-endian addressing orders bytes from the integer's least significant byte. Little-endian addressing orders bytes from the integer's most significant byte.) Table 3-1 lists a small sample set of CPUs, buses, and their endian characteristics:

TABLE 3-1 Byte ordering

Device (CPU or Bus)	Little endian	Big endian
MC680x0 Family of CPUs		x
Intel x86 Family of CPUs	x	
PCI Expansion Bus	x	
Sbus		x

The combinations of byte-ordering relationships are:

TABLE 3-2 Byte-ordering relationships

Big-endian CPU	Big-endian Bus
Big-endian CPU	Little-endian Bus
Little-endian CPU	Little-endian Bus
Little-endian CPU	Big-endian Bus

Virtual memory objects hide any necessary byte swapping from the system software accessing the mapped memory. The byte swapping occurs in software if necessary. However, many computing platforms provide hardware assistance for byte swapping. It is also conceivable that two virtual memory objects could map the same physical bus memory, with one object swapping and the other not.

Data Ordering

Platform independent drivers must also be insulated from data ordering choices made by modern CPUs. Data ordering is a different issue from byte ordering. Byte ordering is the order of bytes (8-bit words) within larger words (16 or 32-bits). Data ordering is the sequence of whole loads and stores (of any size) issued from the processor to and from memory.

Some CPUs reset the order of loads and/or stores to memory. When that memory is mapping a device, undesired side-effects can occur. JPI virtual memory objects can be programmed to force a serialization of all memory access, thus avoiding these unwanted side-effects.

Processor Cache Management

Many processors contain internal caches that can significantly improve the execution time of software. Some processors contain an internal instruction cache which stores the most recently executed instructions. When the processor needs to fetch an instruction it first checks the instruction cache to determine if the instruction is there. Fetching an instruction from the cache is much faster than waiting for the processor to access RAM to fetch the instruction.

Some CPUs contain an internal data cache that holds the most recently accessed data. The data cache operates like an instruction cache but caches data instead of instructions. Before reading data from RAM, the processor checks the data cache to see whether the data is available there. Normally, the functioning of the processor's cache is transparent to applications and drivers. Sometimes, drivers must make sure that the content of the caches and the corresponding information in RAM remain coherent. This may be necessary when information in RAM changes due to a DMA transfer, and the bus hardware does not notify the processor of the change. The cached information is said to be stale or incoherent. To avoid stale or incoherent caches, the cache must be flushed whenever a device performs an operation that can cause the cache to become stale.

Flushing a cache forces the processor to reload the cache from main memory. However, flushing has some well-know adverse side effects. Because it interrupts the execution sequence, for example, flushing decreases performance. It also increases the latency associated with reloading the cache from memory.

In a write-through cache, any data written to the cache is immediately written to RAM; actions by the processor on the cached data range never result in incoherence because whatever is written to the cache is also written through the main memory. The same may not hold true when an external device directly modifies an area of main memory which is also cached. In this case the cache may not be updated concurrently with the modification of the main memory.

With copy-back caching, any data written to the cache is written to RAM only when necessary to make room in the cache for different data accessed more recently, or else when the cache is explicitly flushed. Copy-back caching results in an even greater performance increase but the risk of incoherence with RAM is also greater. Devices which access the RAM directly should ensure that a cache flush is performed for the range of memory they access before attempting to do so to ensure coherency.

The caching methods of the system software are useful in situations where the hardware in a computer system does not ensure coherency between the processor's caches and areas of main memory written by physical devices. For example, if a DMA device writes an area of memory which is currently cached by the processor, incoherence between the processor cache and main memory results unless the bus hardware informs the processor of the incoherence. Devices that require software mechanisms to support coherency with memory and the processor caches are identified in the system device tree. If possible, such devices should inhibit caching by the processor of the memory ranges which they access.

The Cache Management APIs operate on a range of virtual memory and support three modes of caching: inhibited, write-through, and copy-back. APIs exist to return the size and configuration of the CPUs caches.

The system software assigns default cache modes to the known areas of virtual memory. I/O spaces (Fixed, Expansion) and Platform Control are set to inhibited (non-cached) while the remaining areas of virtual memory default to copy-back cache mode.

The Runtime software, including device drivers, may modify the default cache settings to gain a performance advantage. In particular, the AWT framebuffer cache setting is changed to write-through mode.

I/O Operations and Memory

Operating systems must provide services to coordinate the CPU, caches, and memory during I/O operations. Some services are designed to be called once and establish persistent memory settings such as a caching mode. Other services are designed to signal the beginning and end of an individual I/O operation.

The following aspects of I/O management impact these two kinds of I/O coordination services:

- *Memory protection:* The I/O transfer must be consistent with the access restrictions of the specified buffer. For example, writing to ROM or to a read-only range of memory such as a Microkernel text area should be forbidden.
- *Residency:* An I/O operation should not cause page faults that need to be serviced by the Microkernel.

- *Addressability:* The I/O buffer must be addressable to the device performing the I/O transfer. If the device is in fact the CPU (programmed I/O, for example), a virtual buffer address is required. If the device is another bus master such as a DMA controller, a physical address is required. Sometimes both virtual and physical addresses are required during a transfer using programmed and DMA style I/O.
- *Coherent Memory:* Coherency ensures that data being transferred is consistent with all full or partial copies of the data in cache or memory.

The system software provides services that ensure the timely coordination of I/O operations and memory management. These I/O coordination services are provided at a fine granularity to device drivers.

A driver can put together a sequence of small services to coordinate a long-term memory buffer such as a ring buffer or a short-term memory buffer for a one-shot disk read. Drivers that use these I/O coordination services are insulated from the specific quirks of the underlying computing platform, enhancing portability.

Interrupt Management

An interrupt is an asynchronous event generated to request attention from a CPU. An interrupt is generated by an entity called an interrupt *source*. Typical interrupt sources include a CPU itself, a device such as a timer or an expansion bus slot. To a certain extent, a computing platform is characterized by its interrupt architecture. Each motherboard is designed to recognize, route, and dismiss interrupts in a unique manner. Platform families such as a x86-based PCs adhere to standards applicable to just that PC family. Motherboard designs for network computers present a model unique to that platform.

A typical platform interrupt scheme supports the notion of:

- Interrupt Requests
- Interrupt Source Recognition
- Interrupt Source Enable and Disable Modes
- Hardware and Software Dispatch
- Interrupt Acknowledgment and Dismissal
- Interrupt Prioritization.

The following figure illustrates an interrupt model supporting two devices, an interrupt controller, and a CPU. The two devices share a single interrupt request line into the interrupt controller. It is the job of the interrupt controller to prioritize and gate interrupt requests destined for the CPU. The CPU supports a single interrupt request line (IREQ) and one acknowledgment signal (IACK).

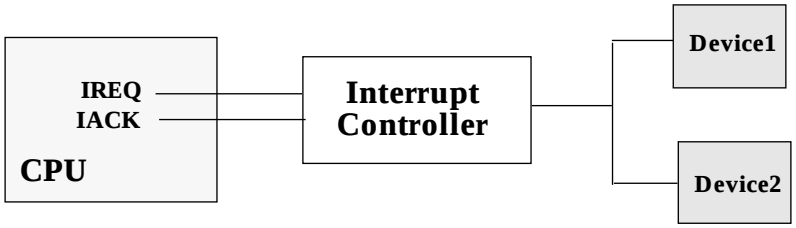


FIGURE 3-8 Simple platform interrupt model

When a device raises its interrupt request signal, the interrupt controller attempts to pass-on this request to the CPU. The CPU is notified of the interrupt request when the interrupt controller raises the CPU's IREQ signal. The CPU begins interrupt processing usually by switching to an interrupt context and then to a software handler. The handler runs and recognizes which device is interrupting typically by reading some register in the interrupt controller. The handler then services the device and returns to the operating system. Either the handler or the OS must acknowledge the interrupt before restoring the CPU to its pre-interrupt execution context.

Each JavaOS for Business platform normalizes this process by presenting a unified, platform independent interrupt management model. To achieve the goal of platform independence, the JPI provides APIs that abstract:

- CPU Interrupt Models
- Bus Interrupt Models
- Device Interrupts
- Interrupt Controllers
- Interrupt Handler Registration and Invocation

The interrupt management model is implemented as a combination of runtime, microkernel, and boot system responsibilities. Each of these layers plays a part in enabling platform independent interrupt processing by device drivers.

JavaOS for Business Interrupts

The system software supports both Java and native interrupt handlers. Java handlers are methods (conforming to the `InterruptHandler` interface) registered by device drivers. These Java interrupt handlers run in the context of a single system interrupt thread. Native interrupt handlers are added by modifying native platform source code. Native handlers run at the CPU's interrupt level associated with the handler.

Abstracting Interrupt Sources

An interrupt is the means by which a device requests attention from software such as a device driver. A device may be embedded within the CPU, attached directly to the CPU's external bus, or to another expansion bus such as PCI. The software represents any device capable of interrupting as an *interrupt source*.

Interrupt sources are related to other interrupt sources. For example, a device on the CPU's bus interrupts the CPU at some CPU-specific level. The CPU's interrupt level is a source of interrupt as is the device itself. When a PCI device generates an interrupt, the signal is propagated over one or more bus bridges until the CPU itself is interrupted.

The topology of interrupt routing and the device topology are similar, but not always identical. Because of the possible differences between device and interrupt topologies, a class of objects separate from devices is required to represent an interrupt source.

The system software creates interrupt source objects to represent the CPU, each CPU interrupt level, and each bus-device combination. The following figure illustrates an interrupt source object for the CPU, two CPU levels, a bus interrupt source, and two device interrupt sources.

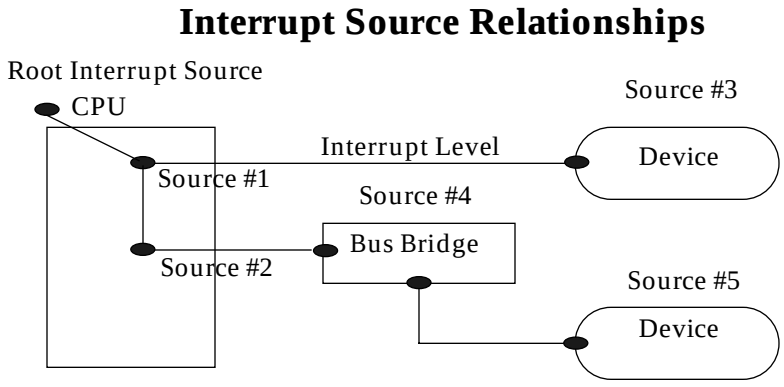


FIGURE 3-9 Interrupt Sources

Interrupt Source Object Organization

The interrupt classes manage the set of known interrupt sources. The set of active and possible interrupt source objects is organized in a hierarchical fashion within the JSD *interrupts* namespace.

Interrupt Source Tree (IST)

The interrupts namespace (also referred to as the Interrupt Source Tree or IST) is pre-created in the JSD on a JavaOS for Business software platform only. The JSD on non-JavaOS for Business platforms such as Windows doesn't include the IST because the underlying OS manages interrupts. (See Chapter 4 for more details on namespaces and database entries.)

On a JavaOS for Business platform, each JSD entry within the interrupt source tree represents a device capable of requesting attention via an interrupt. Each successive tier of the IST represents a finer and finer granularity of interrupt source from CPU (the tree root) to CPU levels to buses (parent entries) and finally to devices (leaf entries).

Representing interrupt sources using a tree that is separate from the device tree has its advantages. Interrupt routing, for example, doesn't always follow the device to bus connectivity. Some platforms require software to route and decode interrupt information, others have sophisticated hardware assist.

Designing device drivers and bus managers to handle all the various combinations of interrupt decode and dispatching logic is a nearly impossible task. The IST acts as a buffer, isolating and protecting portable device driver and bus manager software from the platform's interrupt hardware.

Each platform may require a differently shaped tree to convey the logic and flow of interrupt processing. The IST frees platform designers to use the latest and greatest interrupt controllers without fear that a driver will break. Device drivers never access interrupt controllers associated with a CPU (i.e., a PIC) or a bus bridge controller.

Figure 3-10 shows how the previous example of related interrupt sources is represented in the IST.

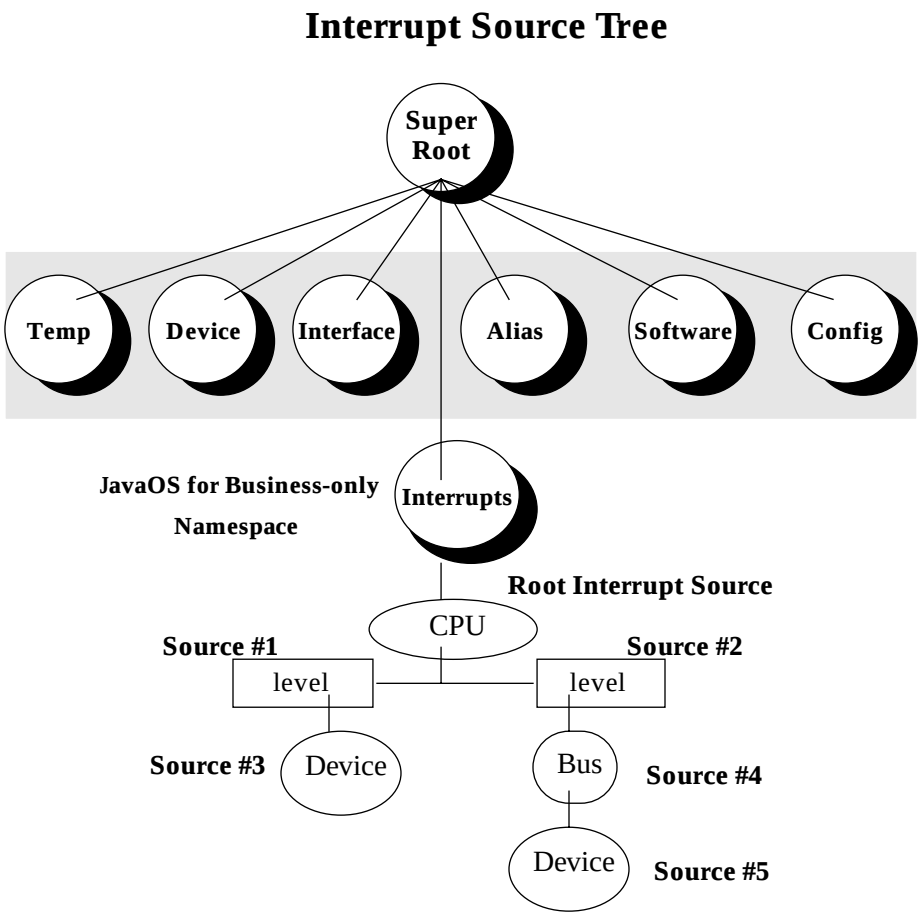


FIGURE 3-10 Interrupt Source Tree in the JSD

Interrupt Source Entries (ISE)

Each interrupt source (whether CPU, bus, or device) is represented using a system database entry called an Interrupt Source Entry (ISE). ISEs are Java objects that are accessed (sometimes concurrently) from both Java and native code.

When creating an ISE to represent a bus, the maximum number of associated child (or device) ISEs must be specified. Pre-creating child “slots” has several advantages for the native code that must access an ISE at hardware interrupt level.

- One advantage for native code is that all memory associated with the object can be easily locked down to prevent page faults. Most Microkernels cannot handle page faults at the interrupt level.
- Secondly, a bus ISE can store references to device ISEs in a pre-created array instead of using the ISE’s linked lists. Using an array within an ISE allows bus managers to number the child devices and use that number as an array index. This is especially important at hardware interrupt level. Indexing into an array using Java Native Interface (JNI) is much simpler than running a linked list.

The following figure illustrates a bus ISE’s array of child ISEs.

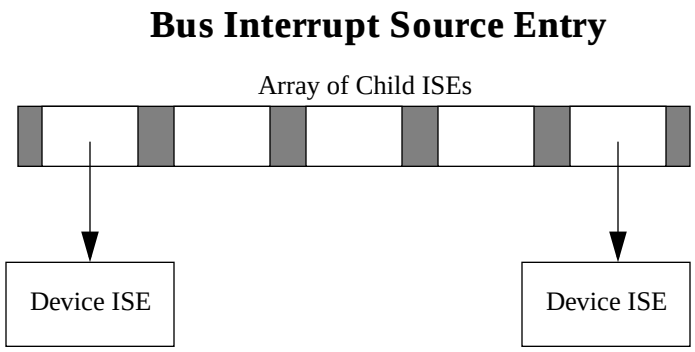


FIGURE 3-11 Bus Interrupt Source Entry

IST Construction

The IST is built dynamically as the system software initializes the platform. First, the platform manager creates the interrupts namespace root, and installs itself as the manager of this namespace. (See Chapter 4, JSD regarding namespace managers.)

Next, the platform manager creates a single root CPU Interrupt Source Entry (ISE), and then proceeds to create a child ISE for each CPU interrupt level (capable of interrupting on this platform).

IST Topology and Management

The Platform Manager, each bus manager, and each driver interact with ISEs and the IST in some manner.

The Platform Manager creates the *interrupts* namespace, and then creates a single CPU object and multiple CPU level objects. Each JavaOS for Business platform comes with built-in handlers (native and Java) for each CPU level.

Bus managers, including the platform's CPU bus manager (i.e. Platform Manager) maintain the IST on behalf of drivers and other bus managers. For each device under the control of a bus manager, at least one entry in the IST is created to represent that device's interrupt source.

Drivers construct interrupt source objects and then ask the appropriate bus manager to install it in the IST.

The following objects illustrate interrupt source management at various levels of the IST.

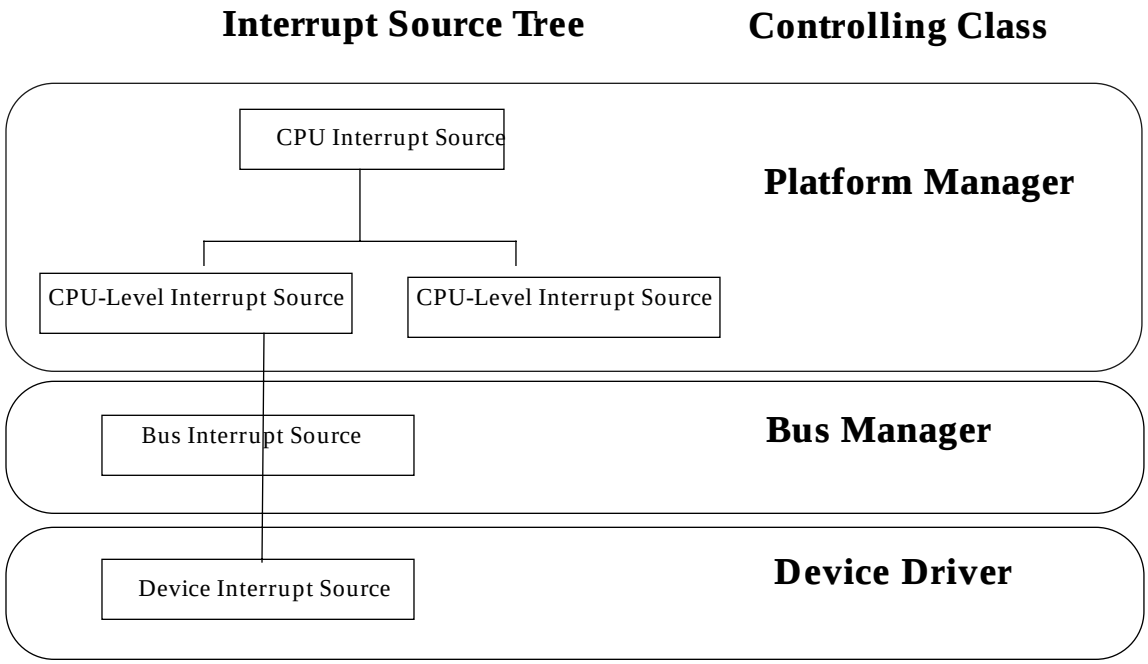


FIGURE 3-12 IST Management

It is up to the Bus Manager to decide where to install the Interrupt Source Entry Object in the IST. For example a Bus Manager could install an ISE for a device directly under a CPU-Level Interrupt Source Entry as shown in the previous figure.

Device and Interrupt Namespace Coordination

As the device manager matches devices with device drivers and buses with bus managers, the tree grows and handlers are installed. The platform manager and each bus manager that is activated grows the tree by adding child interrupt source entries. Each new child ISE is cross-referenced with a device entry in the device namespace, using a JSD property. The following figure illustrates how the device and interrupt namespaces are cross-referenced.

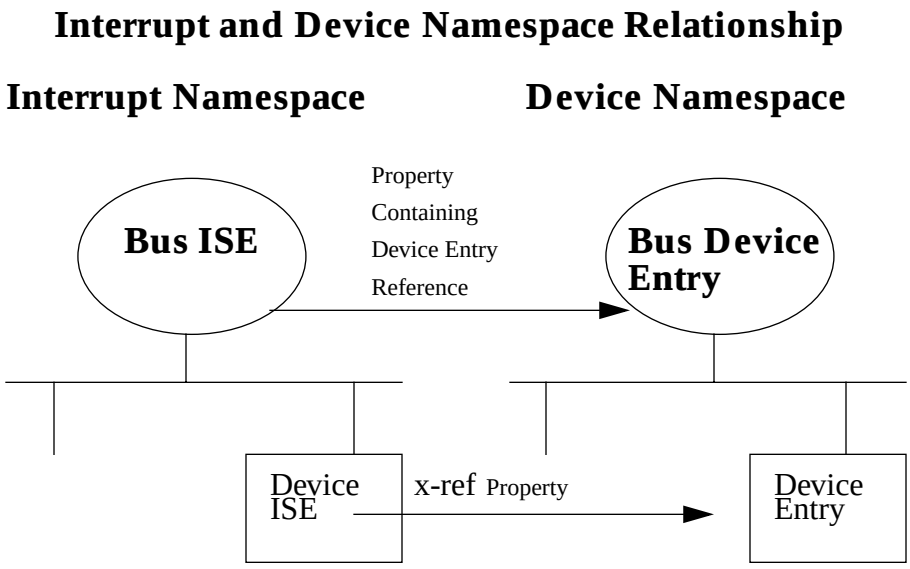


FIGURE 3-13 Interrupt and Device Namespace Cross-Referencing

Interrupt Code Registration

Four kinds of code can be registered with an interrupt source entry.

1. An interrupt handler executes in response to an interrupt. Its job is to satisfy the device's request for attention, secure any realtime data, and return control to the operating system as soon as possible.

2. An interrupt enabler places the device in a state where interrupts are possible or *unmasked*. A CPU interrupt enabler manages all interrupts sharing the CPU interrupt controller (internal or external). A CPU level enabler unmasks just a single level of interrupt. A bus or device enabler unmasks just those interrupts from a bus or device.

3. An interrupt disabler places the device in a state where interrupts are impossible or masked.

4. An interrupt acknowledger communicates to hardware the fact that the OS and/or device driver has handled (or acknowledged) the interrupt. Acknowledgment can take place before or after a handler is actually dispatched.

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. The three kinds of interrupt processing are:

1. First-level realtime native interrupt processing

2. Second-level deferred native interrupt processing

3. Third-level deferred Java interrupt processing

The Microkernel oversees the first two levels of interrupt processing. The third (Java-centric) level of interrupt processing is supported by the JVM using Java threads.

Each kind of handler associated with an interrupt processing level runs in its own special interrupt execution context. A single interrupt source can have none, any, or all of these handler kinds assigned and working as a team to process interrupts.

Each level of interrupt processing can communicate state and data to any other level using the interrupt source entry as a common data exchange point.

The latter two of these interrupt processing levels are *deferred*. A deferred interrupt level is desirable for non-realtime processing, and is triggered from native code only.

First Level Handler

The first kind of interrupt handler is a native interrupt handler that executes at CPU interrupt level. This handler is composed of native code that may or may not have been compiled from C. This handler obeys C calling conventions defined for the processor, even if the handler is composed of assembly language.

Upon entry to the first-level handler, the Microkernel has already saved processor registers and may have switched to a separate interrupt stack. The choice of which stack to run first-level handlers on is up to the Microkernel. Many Microkernels dedicate an interrupt stack per-CPU for this purpose. Others execute the handler on the stack of the current thread, unwinding the stack upon the completion of the processing.

The code in this kind of handler runs when invoked by the Microkernel. Interrupt level execution context provides the following support services to the first-level handler. A first-level interrupt handler can use JNI (Java Native Interface) to:

- Read and write data within the interrupt source object.
- Traverse the IST and subsequently dispatch other first-level interrupt handlers.
- Signal the object, so that waiting threads at second, third, or main execution levels run.

A first-level handler 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 realtime needs of a device, a first-level interrupt handler can cause a second or third-level handler to be queued by the Microkernel. A first-level handler can choose to pass state and/or data to deferred second and third level handlers through the interrupt source object using JNI to get and set data within the object.

The handler C function prototype matches that of an interrupt source object native method prototype with a single long integer parameter that contains the current time since boot in microseconds. The handler returns a long integer that signals whether the interrupt was processed by the handler. An example of a native CPU level interrupt handler is:

```
typedef long firstLevelHandler(void * ise, int64_t when);
```

Note: Multiple first-level interrupt handlers can be executing simultaneously, each at a different level. This is true for single CPU's and also in an SMP system. In an SMP system however, the Microkernel serializes the execution of each interrupt level so that two or more CPU's don't attempt to execute the same handler simultaneously.

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 second kind of interrupt handler runs in the context of a native interrupt thread. This handler is also composed of native code that obeys C calling conventions, and is structured as a native method with a single parameter.

Like a first-level handler, a second-level handler has a limited number of support services at its disposal. Native handlers (first and second levels) may only use JNI to get and set ISE data and to invoke other native methods associated with the same ISE.

A second level interrupt handler is queued to run under two circumstances. A first-level interrupt handler can queue a second-level handler, or if no first-level handler exists, the Microkernel will automatically queue the second-level handler in response to an interrupt. The current implementation of the system software does not support second level interrupt handlers.

An example of a native second-level interrupt handler is:

```
typedef long secondLevelHandler(void * ise, int64_t when);
```

The native interrupt thread is created by the Microkernel during the second-level handler registration process. The stack allocated for a native thread is at least a page in length.

Second-level interrupt handlers run after first-level interrupt handlers, and may preempt third-level handlers and other Java or native threads.

Third Level Handler

A Java interrupt handler runs in the context of any Java thread, including a pre-created Java system thread, and therefore may use the full resources of the language, the system software, and JDK.

A third-level interrupt handler is queued to run under the following circumstances:

- 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.
- 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 Java based interrupt handler method in the `DeviceInterruptSource` class is defined below and is overridden by the derived specific device driver classes to do some useful work.

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

Java interfaces are defined for each kind of interrupt management code. Details on the interfaces are available in a separate file as part of this document and are explained in Chapter 9, JavaOS for Business Classes.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on to the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

Interrupt Level Management

First-level interrupt handlers are necessary when a device may lose data if not attended to in a timely fashion. The duration of a first-level interrupt handler should be measured in microseconds.

Second-level interrupt handlers are useful when doing extended realtime processing such as is necessary with multi-media applications.

Third-level interrupt handlers are useful when doing non-realtime processing, such as mouse and keyboard event handling. Third-level interrupt handlers may experience sporadic latencies due to the necessity to synchronize with the virtual machine's garbage collector. If these latencies hinder device operations, a second-level interrupt handler should be used.

Synchronizing Interrupt Handlers

With interrupt handling comes the problem of synchronization. The system software allows a driver non-interrupt level code to synchronize with all three levels of interrupt processing.

Each thread that executes second and third level handlers acquires the Java monitor associated with the ISE before executing the handler. (A Java monitor is a synchronization mechanism associated with all Java objects.) By acquiring the ISE's monitor, a driver can prevent a second or third-level handler from running . If an interrupt handler is already running, the driver blocks until the handler releases the monitor. If the monitor is free, the driver acquires the monitor and blocks any subsequent monitor acquisition attempt by a handler until releasing the monitor.

Synchronizing with code not executing within a thread context (i.e., first-level handlers) requires the driver to enable and disable the interrupt source itself. Each interrupt source object implements an interface containing enable and disable methods for this purpose.

The following figure illustrates the use of Java monitors in interrupt processing.

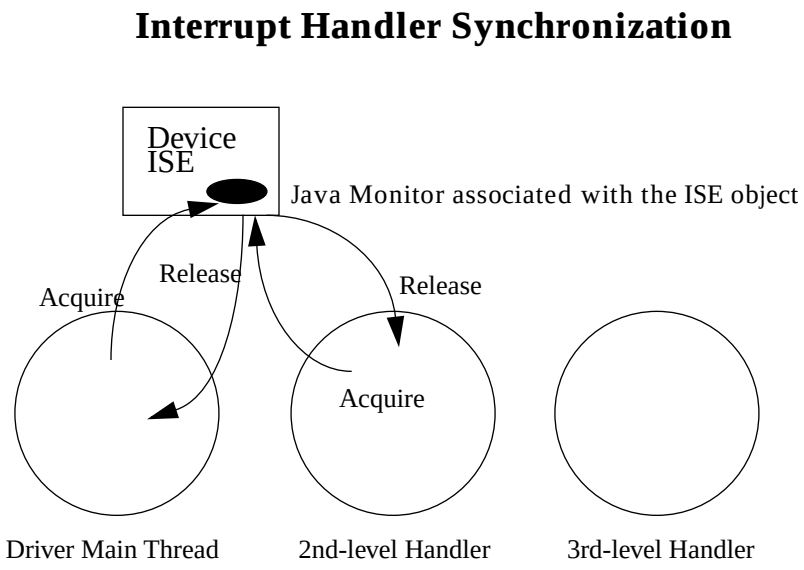


FIGURE 3-14 Synchronizing Interrupt Handlers

Queuing a Deferred Interrupt Handler

Native first-level and second-level interrupt handlers can choose to defer work to higher interrupt levels on a per-interrupt basis. A simple mechanism is required to allow drivers to defer work.

To defer work from a lower interrupt-level to a higher-interrupt level, an interrupt handler merely notifies the current interrupt source object. Notifying the interrupt source object, causes the virtual machine to wake-up threads waiting on the interrupt.

For example, a third-level interrupt handler can run in the context of a pre-created Java System Thread or in the context of any other thread.

The waiting thread maintains a loop something like:

```
public void run {
                  while (true) {
                      try {
                          long when = waitForNextInterrupt();
                          handleThirdLevelInterrupt(when);
                      } catch (Throwable e) {
                          ...
                      }
                  }
}
```

The following figure illustrates the concept of deferred interrupt handling:

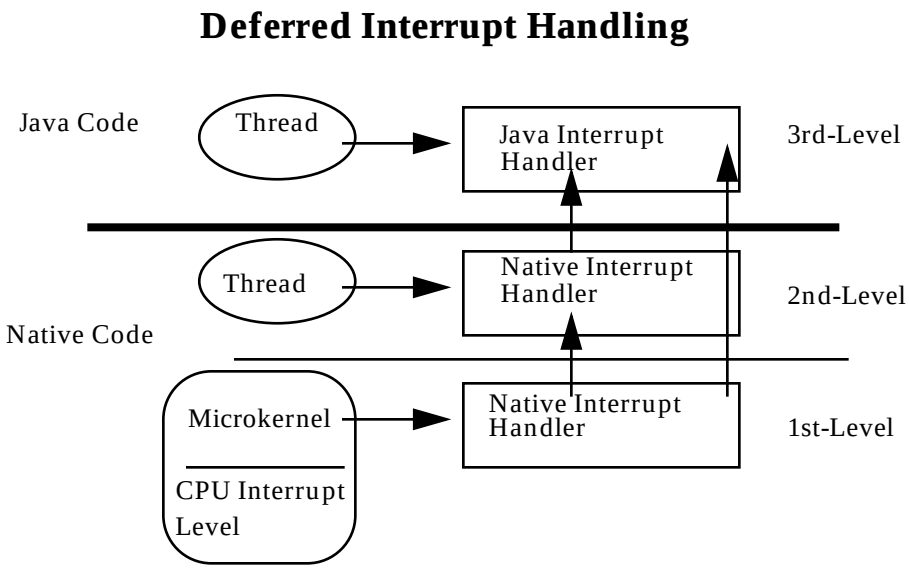


FIGURE 3-15 Deferred Interrupt Handling

Interrupt Dispatching

Dispatching an interrupt handler requires access to the IST. The IST is made accessible to the native methods of the system software `CpuLevelInterruptSource` class during platform initialization.

The native portion of the interrupt class makes sure that each entry in the tree (a Java object) is locked down in the Java heap for the duration of its lifetime.

Interrupt Dispatcher

The interrupt dispatcher is a component of native code that executes first and second-level interrupt handlers. The interrupt dispatcher is layered upon the Microkernel, and actually registers itself as a handler of all interrupt vectors that the Microkernel exports.

Figure 3-16 shows the interrupt dispatcher layered upon the Microkernel.

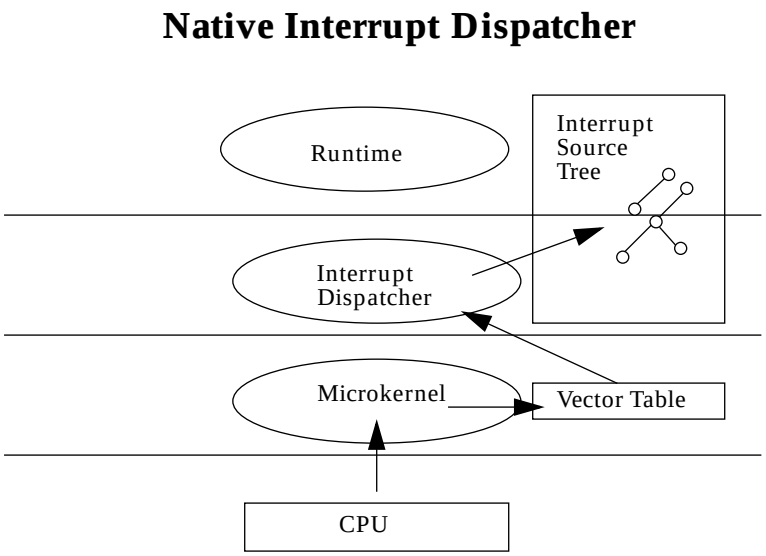


FIGURE 3-16 Interrupt Dispatcher

The Microkernel has no knowledge of the IST. The interrupt dispatcher presents itself to the Microkernel as the sole handler of all interrupts. This design requires little of an underlying Microkernel besides a simple interface to install and remove CPU vector interrupt handlers.

In this example, the Microkernel has the following C interface supporting the interrupt dispatcher:

```
void (*native_handler)(int level);
void set_native_intr_handler(int level, native_handler func);
```

The `level` parameter designates a CPU level (or vector). The `func` parameter designates a native interrupt handler. Passing in a null handler results in the Microkernel removing the handler. The Microkernel passes the current interrupt level to the native handler as the sole parameter.

Bus and Device Interrupt Handling

Each bus and device has an associated interrupt handler. Bus interrupt handlers perform a special decoding function that actually augments the handler invocation logic used by the interrupt dispatcher.

The job of the bus handler is to determine which device interrupt handler should be invoked next. The device interrupt handler is invoked by the bus interrupt handler using JNI. In the case of layered buses, multiple bus interrupt handlers may be invoked to decode an interrupt. Eventually a device handler is invoked that actually processes the device's interrupt.

Bus and device interrupt handlers can exist at all levels of interrupt processing depending upon platform requirements and performance considerations. The process is always the same however. Bus interrupt handlers determine the source of the interrupt on a particular bus. Device interrupt handlers process the interrupt. The only difference at each level is the execution context: interrupt or thread, native or Java.

The following figure illustrates bus and device interrupt handlers.

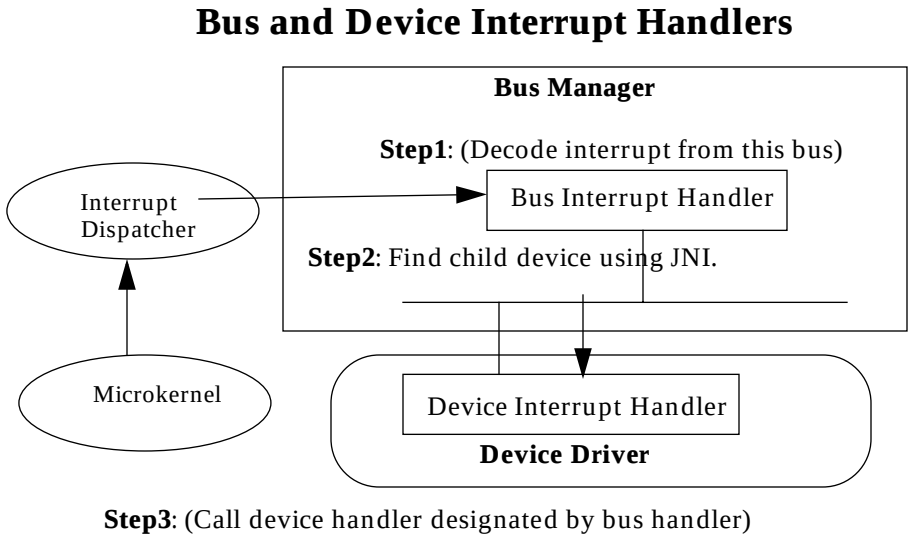


FIGURE 3-17 Two Types of Interrupt Handlers

Class Hierarchy

The following figure illustrates the InterruptSource class hierarchy.

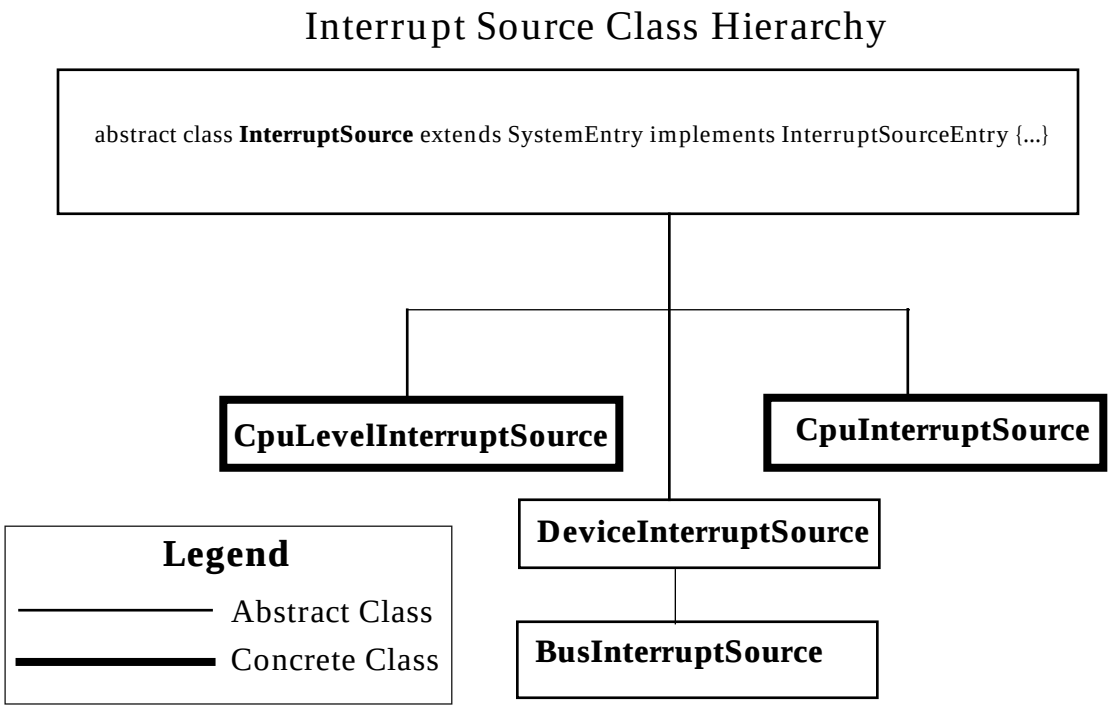


FIGURE 3-18 InterruptSource Class Hierarchy

The Boot System

The Boot System carries out two important interrupt functions. It:

- Provides bus and device specific interrupt topology description to the microkernel. These interrupt descriptions conform to the OpenBoot 1275 Standards published for each kind of bus and device. These descriptions take the form of device properties attached to the device tree. For example, for each PCI device on the platform an interrupt property defines the contents of the Interrupt Pin register from the Configuration Space header. (See Chapter 2, JDI and Chapter 7, JBI for a more detailed description of the device tree and its properties)
- Places all devices in a disabled or inhibited interrupt state upon invocation of the microkernel.

The Microkernel

The Microkernel provides the following interrupt services. It:

- Provides a normalized interrupt topology description to the runtime. The microkernel transforms the bus and device specific interrupt information into a bus and device independent form for use by Java-based drivers executing in the Runtime.
- Places all devices in an enabled interrupt state upon invocation of the runtime.
- Installs default native code handlers for all interrupt sources. It's the job of these default handlers to recognize the exact source of the interrupt. Once the source is located, the Microkernel can invoke either a native or Java-based handler to service the interrupt.
- Provides C-based native interrupt handler registration API.
- Provides C-based native interrupt source disable API.
- Provides C-based native interrupt source enable API.

The Runtime

The Runtime has two interrupt functions. It:

- Presents a Java-object based view of the interrupt sources to candidate drivers.
- Provides Java-based handler registration APIs.

DMA Management

DMA abstraction is essential to achieving platform independent device drivers. The following categories of DMA are recognized by JavaOS for Business:

- *Device DMA.* If the device itself is capable of acting as a true bus master, the driver programs the DMA's address and count registers *directly in the device.* Devices on most SPARC-based platforms use this form of DMA.
- *Platform DMA.* This flavor of DMA uses a centralized DMA engine on the platform's motherboard. This centralized DMA device is a shared resource with many channels available for long-term or short-term allocation. For each DMA transfer the driver must program both the device and the DMA channel to initiate a transfer. A set of platform DMA APIs allows a driver to use a platform DMA channel in a portable fashion.

In both platform and device DMA, a driver uses one or more physical addresses to initiate the transfers. These two DMA transfer types are illustrated in the following figure:

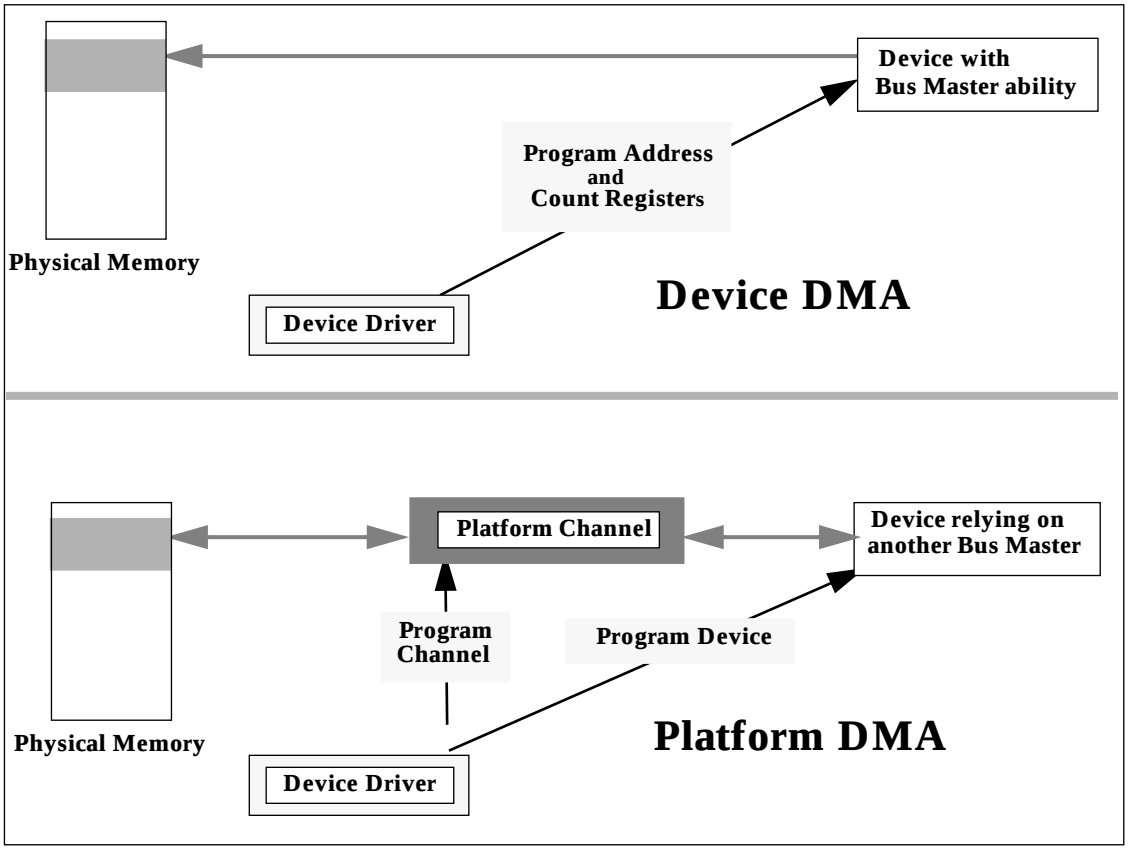


FIGURE 3-19 Device DMA and Platform DMA

On many platforms, a physical address has a one-to-one relationship with a virtual address. From the driver's point of view however, all DMA is accomplished using physical addresses.

The platform interface contains APIs that allow drivers to use a platform supplied DMA controller. A shared platform DMA device is described by the following set of attributes:

- Lowest and highest physical address the DMA controller can handle.
- Maximum transfer size.
- Minimum transfer size.
- Address alignment considerations.

JavaOS for Business Memory and DMA Classes

Figure 3-20 illustrates the memory and DMA class hierarchy.

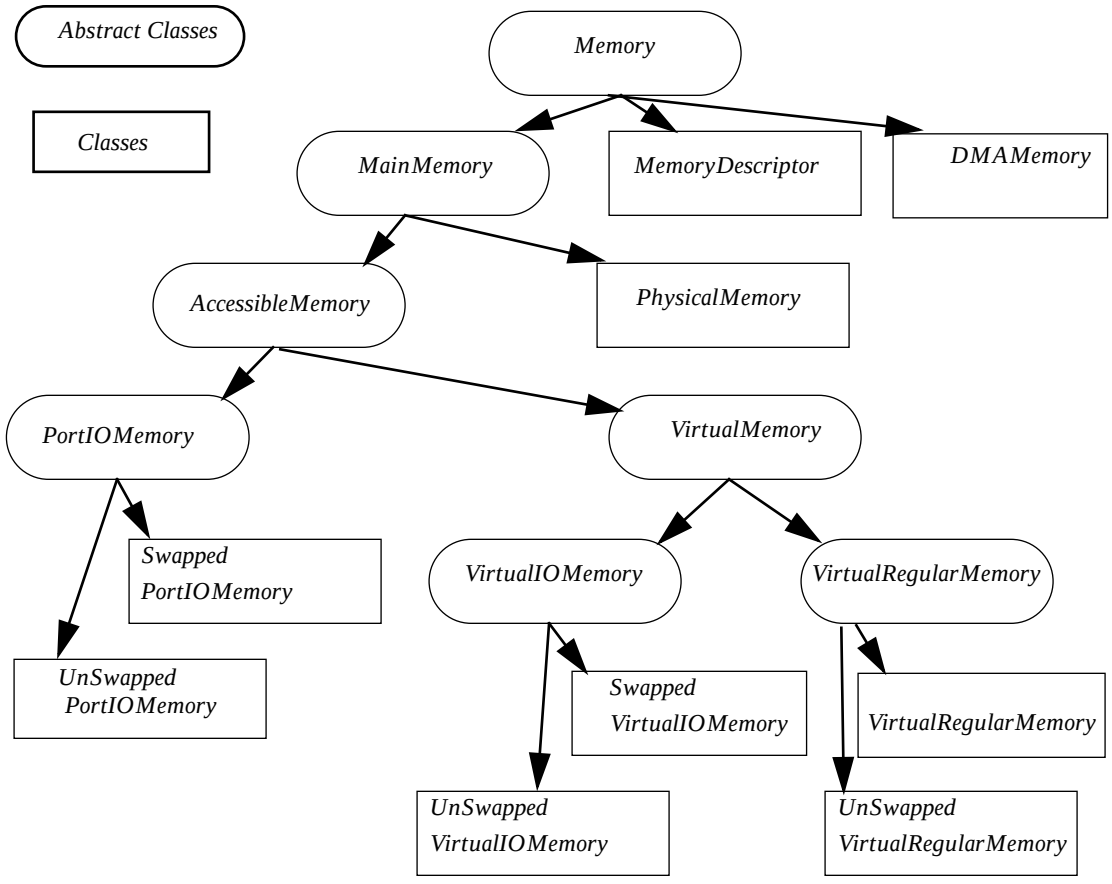


FIGURE 3-20 JavaOS for Business Memory and DMA Classes Hierarchy

Memory Abstract Class

At the highest level, each of the Memory and DMA classes is ultimately a subclass of the `Memory` abstract class, which has attributes such as base address, length, and constraints (of both access and allocation).

Below the Memory superclass, memory is one of three class types:

- `MainMemory`, which is accessible by a CPU (but not always directly, as in the case of physical memory),
- `Memory Descriptor`, which is a class used as a token to represent memory, or
- `DMAMemory`, which is accessed externally to the CPU via DMA.

MainMemory Abstract Class

The `MainMemory` abstract class specifies abstract methods for managing caching.

These abstract cache management methods are implemented in `PhysicalMemory`, `VirtualMemory`, and `PortIOMemory`. It is important to note that cache management is highly platform-dependent, and thus each of these implementations ultimately relies on invoking the Microkernel to flush caches, set the cache mode, etc.

`MainMemory` is either the abstract class `AccessibleMemory`, which represents all memory that can be directly accessed by a CPU, or the `PhysicalMemory` class, which represents physical memory and may not always be directly accessed by a CPU.

MemoryDescriptor Class

Having a non-abstract and hence instantiable subclass 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`.

DMAMemory Class

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

The Microkernel ultimately sets up DMA mappings.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 physical DMA addresses. The addresses are simply DMA addresses. The mappings are transparent to DMAMemory and maintained entirely by the Microkernel.

AccessibleMemory Abstract Class

AccessibleMemory is what is given to drivers by bus managers, so that they can always access memory in a platform-independent manner, allowing them to be portable across all platforms. AccessibleMemory thus defines platform-independent abstract methods, such as setByte and getByte. Drivers should only use the methods abstracted in AccessibleMemory, or methods abstracted or implemented by its super-classes MainMemory, and Memory. PhysicalMemory and DMAMemory and those classes below AccessibleMemory should only be accessed by bus managers, which understand platform specifics and shield drivers from the underlying hardware.

AccessibleMemory is either accessed by a CPU issuing normal load and store instructions (VirtualMemory) or by a CPU issuing special I/O port instructions (PortIOMemory).

PhysicalMemory Class

PhysicalMemory is not accessible because (even on systems that do not have MMU), CPUs are really accessing virtual memory that happens to be mapped one-to-one to physical memory. This allows the system software to handle systems with and without MMUs in the same general way. Since the PhysicalMemory class is a sub-class of the abstract class MainMemory, PhysicalMemory allows implementations of the cache management methods abstracted in MainMemory.

The PhysicalMemory class lets a bus manager specify a range of physical addresses, device registers, interrupts, and so on. Since PhysicalMemory cannot be accessed, it is not useful by itself. However there are methods of VirtualMemory and DMAMemory for mapping this physical memory into virtual memory or DMA, respectively.

PortIOMemory Abstract Class

The other type of AccessibleMemory besides VirtualMemory is PortIOMemory, which is intended for x86 CPUs issuing special port I/O instructions. The PortIOMemory abstract class itself only implements the cache management operations abstracted by MainMemory.

VirtualMemory Abstract Class

The `VirtualMemory` abstract class contains methods for setting up mappings to `PhysicalMemory` objects (mappings ultimately set up by the Microkernel). As mentioned above, even on systems with no MMUs, `VirtualMemory` objects are mapped one-to-one to `PhysicalMemory` objects. The `VirtualMemory` class also has a method for allocating a specific amount of virtual memory (also ultimately done by the Microkernel), without specifying any particular physical memory. There is even a method which then allows a system to obtain an array of the underlying `PhysicalMemory` objects for a `VirtualMemory` object, which is especially useful if `PhysicalMemory` objects were not passed to construct the `VirtualMemory` object in the first place.

It is possible for a bus manager to set up DMA to virtual memory, since, given a `VirtualMemory` object, the array of underlying `PhysicalMemory` objects can be obtained, and then passed to a method of `DMAMemory` to set up DMA. The reverse is also possible, where, after a `DMAMemory` object is constructed, an array of the underlying `PhysicalMemory` objects can be obtained, and then used to construct a `VirtualMemory` object, which is accessible.

The mappings from virtual to physical must be locked during DMA, so the `VirtualMemory` class also supports `lock` and `unlock` methods. Otherwise, the Microkernel might replace one underlying physical page with another one, after the original physical page has been mapped into DMA hardware. There is even a `lockContiguous` method, which ensures that mapped physical pages are contiguous; this method is necessary on platforms that do not have a mechanism, such as an IOMMU, for doing scatter-gather. The bus manager specific to the platform must decide what methods are needed.

The `VirtualMemory` abstract class also implements cache management methods (ultimately performed by the Microkernel), which are abstracted by `MainMemory`. These methods also are useful when setting up DMA, depending on the coherency characteristics of the platform and the platform specifics known by the bus manager.

SwappedPortIOMemory and UnSWappedPortIOMemory Classes

The bulk of the support for `PortIOMemory` is in its two subclasses, `SwappedPortIOMemory` and `UnSwappedPortIOMemory`, which each implement all the memory access methods abstracted by `AccessibleMemory`. As the names imply, one class performs byte-swapping of the data written (to support situations where, for example, the CPU is big-endian and the bus architecure is little-endian), and the other does not. It is up to the bus manager to decide which class to instantiate.

Having two classes, one for swapped data and one for unswapped data, means that it is not necessary to have a conditional statement that checks whether memory is byte-swapped in each of the native methods.

VirtualIOMemory and VirtualRegularMemory Abstract Classes

`VirtualMemory` is either mapped to I/O space (`VirtualIOMemory`), or it is not (`VirtualRegularMemory`). The reason for the two different abstract classes is that on some platforms, access to I/O space may also require operations to flush out the pipeline.

Neither `VirtualIOMemory` and `VirtualRegularMemory` currently provide functionality by themselves, but leave it subclasses to provide functionality. These two classes are included in the event that subclass functionality develops and a higher level of abstraction is required.

SwappedVirtualIOMemory and UnSwappedVirtualIOmemory Classes

Like `PortIOMemory`, `VirtualIOMemory` has two subclasses, `SwappedVirtualIOMemory` and `UnSwappedVirtualIOMemory`, each of which implements all the memory access methods abstracted by `AccessibleMemory`.

SwappedVirtualRegularMemory and UnSwappedVirtualRegularMemory Classes

Like `PortIOMemory`, `VirtualRegularMemory` has two subclasses, `SwappedVirtualRegularMemory` and `UnSwappedVirtualRegularMemory`, each of which implements all the memory access methods abstracted by `AccessibleMemory`.

Addresses

DMA memory, physical memory, and virtual memory are all addressed by addresses from the same `Address` abstract class. Since there are significant examples of systems that have addresses with non-power-of-two number of bits, the `Address` class supports addresses with any number of bits between 1 and 64.

Unsigned Values Representing Addresses

Internally to the class, an `Address` object is represented as an unsigned value, along with an indication of the number of bits. Since Java does not have unsigned arithmetic as part of its foundation classes, it was necessary to develop some simple efficient algorithms for addition and comparison of addresses.

The `Address` abstract class is abstract mainly because it is most efficient to use different algorithms for unsigned arithmetic, depending upon the number of bits used to represent the address value. There are two sub-classes, `Address32` for addresses up to 32 bits, and `Address64`, for addresses up to 64 bits.

Platform Dependence and Platform Independence

When a bus manager constructs any of the Memory and DMA classes, it must provide a base Address and a length (also represented as an `Address`). To do this, it first must know the number of bits for that the specific type of address on the specific platform (bus managers know this platform-specific information). It then constructs each of the Address objects.

Drivers, on the other hand, always access `AccessibleMemory` objects that have been constructed by bus mangers. Drivers do so by providing an offset from the base of the `AccessibleMemory` object. So that drivers can be readily independent of platform specifics, such as the number of bits in a particular address, each memory access method of `AccessibleMemory` specifies the offset as an integer. However, each `AccessibleMemory` object must be smaller than 2GB.

For simplicity, once an address is passed to the Microkernel, it is represented (in native methods) simply as a 64-bit unsigned integer.

Virtual Address Spaces

The system software has only one virtual address space, which it uses to take care of all memory allocation for Java programs.

Use of Virtual Address Space ID

When VirtualMemory objects are constructed, the virtual address space ID is one of the parameters of the constructor. This ID is eventually provided to the Microkernel when the Microkernel is requested to map the virtual memory.

Only the Microkernel assigns semantics to the value of the virtual address space ID.

Interfaces

Java interfaces are defined for each kind of interrupt management code, and are explained in Chapter 9, JavaOS for Business Classes.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

JDI and JPI Interfaces

The Memory classes which the JDI uses to interface with platforms are explained in Chapter 9, JavaOS for Business Classes.

Java interfaces are defined for each kind of interrupt management code, and are explained in Chapter 9, JavaOS for Business Classes.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on to the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

These classes are platform independent, and can thus be imported by platform-independent JavaOS for Business Java code, such as drivers. Since they are platform-independent, these classes do not depend on any of the platform-dependent memory classes in the `sun.javaos.memory` package.

Unlike their JDI counterparts, the Memory and DMA classes in the JPI set of the `sun.javaos.memory package` are platform-dependent. These classes are primarily used by platform-dependent Java code in the system software, such as bus managers, to extend the platform-independent classes of `javax.system.memory`. These classes are also documented in Chapter 9, JavaOS for Business Classes.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on to the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

Microkernel Interfaces

The interfaces to the Microkernel consist entirely of the set of native methods of the various Java classes. Of these two Java class packages, `javax.system.memory` and `sun.javaos.memory`, only the latter has native methods and these are specified in Chapter 9, JavaOS for Business Classes.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on to the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

4

Java System Database (JSD)

Overview

This chapter describes the Java System Database (JSD), as well as several related programs. It presents information about the following topics:

- Overview
- JSD Architecture
- The Server JSD
- Database Schema
- Entry operations
- Entry events
- Transactions
- Navigation
- Class/Interface Summary
- JavaOS Configuration Tool (JCT)

JavaOS for Business uses a layered architecture to provide the smallest, fastest, stand-alone Java programming environment available. Its layered architecture means that segments can be updated independently.

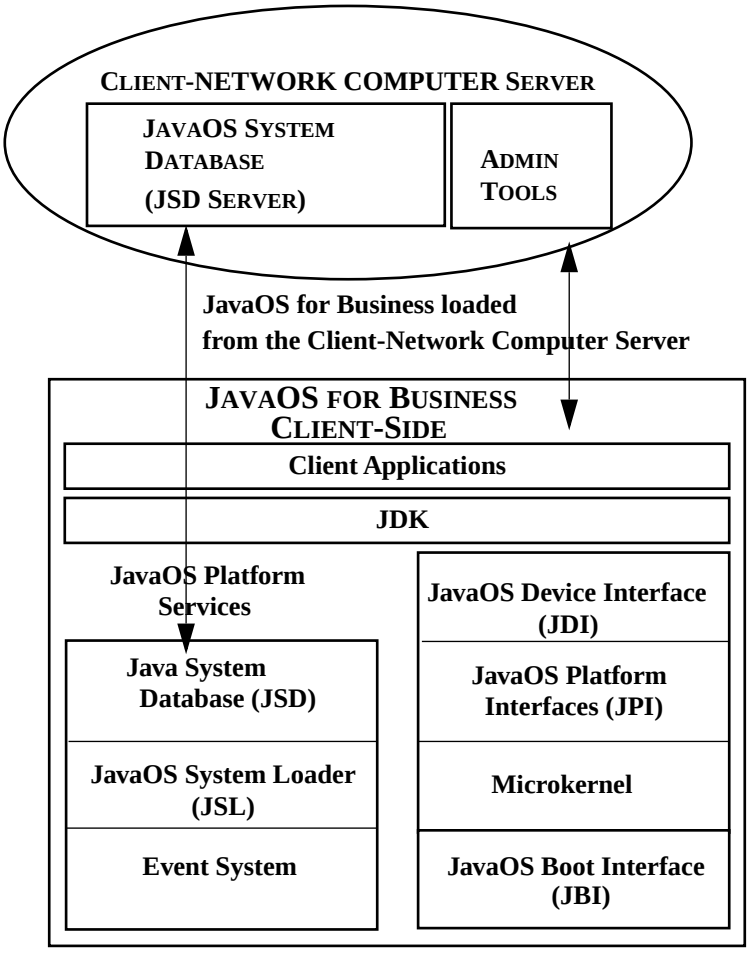


FIGURE 4-1 JavaOS for Business Client-NC Server Overview

The software was designed as a client/server architecture with a series of devices, including a network computer, acting as thin clients. To facilitate this, the software uses a database approach to handle communication between the client(s) and the server(s). The JSD is used by the JavaOS Device Interface, as shown in Figure 4-2, for configuration information.

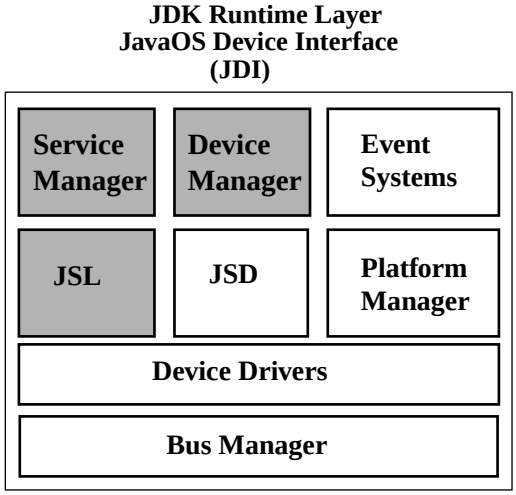


FIGURE 4-2 JSD Placement within JDI

The JSD manages the software configuration information. It lets operating system services, applications, and Java Developer Kit (JDK) components store and retrieve configuration information on all Java platforms.

This configuration information describes what devices are present, what system software services are installed, what user and group attributes have been selected, and any application-specific information required. Each entry in the JSD may have properties to describe (or associate information with) the entry. The standard JDK properties are still accessible outside the JSD.

Important features of the JSD include:

- Client/server implementation (on-client caching and local execution)
- Remote management capabilities (ease of system administration)
- Transaction-based access
- Small memory footprint (under 200 K)
- Event notification
- Persistent storage (standard interface available for FCS)
- Tree-based hierarchy of database entries (smart client storage and retrieval)

The JSD is designed to be populated dynamically during platform start-up using a flexible interface which obtains data from files, the network, and the host OS, as illustrated in Figure 4-3.

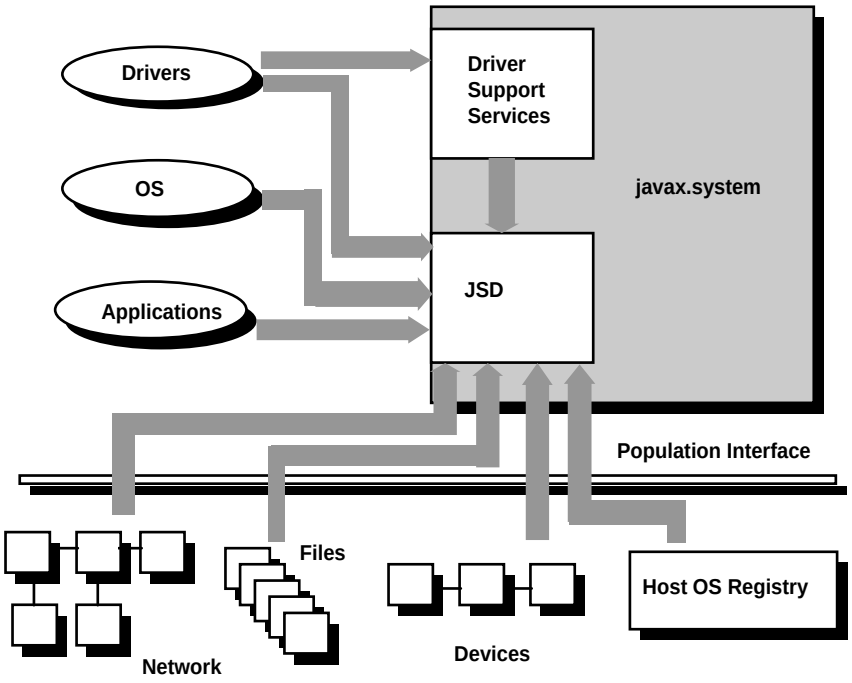


FIGURE 4-3 JSD Overview

The JSD is key to implementing the system's goal of providing a platform-independent, device-independent facility for managing a wide variety of thin client terminals and their need to communicate with a variety of other devices.

It exists in two parallel but slightly different versions: a server side and a client side.

On the server side, a master set of configuration and communications data is kept continually updated with network, software, device, and user information from all platforms.

On the client side, the local version of the JSD serves as a configuration and communications hub. It retrieves, stores, and publishes configuration (as well as operating system) information. Once retrieved, data can be broadcast back or published to requesting sources. It serves as a registry service for a device such as a network computer as long as the device is on, but is restored each time the machine is rebooted. (See the terms *persistence* and *transience* later in this chapter.)

The local populating component or a client/server architecture handles persistent database information. The JSD supports an Event System which can be used to maintain data coherence after the initial population of the database.

While the Java System Database stores and retrieves configuration information on behalf of applications and system services (such as device drivers), it does not dictate how that information is originally acquired or interpreted. The JSD is a unified configuration repository—not, strictly speaking, a configuration manager.

Persistence and Transience

The JSD is designed to take advantage of two key characteristics in a client-server environment that includes network computers and other devices that are routinely turned on and off by users or applications. Under this design, servers are assumed to have *persistent* information (data that does not disappear when the device loses power). What this means to the JSD is that database entries remain accessible to the system whenever it has power, and can be updated whenever the server is running.

However, clients are assumed to have both persistent and *transient* information (data that disappears when the device's power is turned off). The client's persistent information is backed up by the server, and restored when the user logs in or the client machine is rebooted. The client machine may also be able to download any other information it needs from the server, including user profiles, applications, and access to output devices, but when the power is turned off all but its most basic machine identification data is lost.

JSD Architecture

The JSD is designed to provide sufficient configuration management data to ensure interoperability among heterogeneous NC clients and servers. It also provides an infrastructure to let developers integrate their applications into an NC environment. Its design lets a user move from device to device without losing their personal virtual desktop environment.

The JSD has a predefined namespace structure (schema), and includes access methods for creating, modifying, and deleting entries. The registry can be searched or queried and an event model is implemented to notify listeners of changes to the registry. Session management parameters let the state be saved and restored, ensuring desktop persistence for users moving from one system to another.

It is important to point out that the JSD is not a generic directory service that encompasses existing enterprise-wide directory services such as the Novell Directory Service or the Lightweight Directory Access Protocol (LDAP). The software does not implement a naming and directory service but instead relies on service providers such as LDAP, NIS, or DNS to implement such a service.

The JSD is optimized for thin client configurations and provides needed functions that are not in LDAP or JNDI, such as event notifications, atomic updates, and profile coalescing. The JSD provides a small footprint implementation with minimal configuration service for low-memory network computers.

The Server JSD

The server JSD uses a two-tier model (see Figure 4-4).

The Two-tier model has the following key advantages:

- *Simple clients.* The model pushes entry management complexity to the server. Clients remain thin objects.
- *A single Client/Server protocol.* The client only has to use a single protocol to communicate with the server, which is optimized for low-memory footprint and real-time environments. This protocol can be customized to specific configuration requirements.
- *Cached and ready to use coalesced entries.* When a large number of clients talk directly to an external (potentially remote) directory service, processing the entries can produce significant bottlenecks on both the server and the client. The JSD server, however, can act as a middle layer and enable caching of entry information in an already pre-processed format ready to be sent to the client. The server can also act as a filtering agent to reduce the number of requests to a potentially high-overhead directory service.
- *Reads are always local.* Directory operations have a predictable behavior for configuration parameters such as interrupt levels and device information, guaranteeing the correct functioning and performance of the system.
- *Support for multiple-directory services.* In designing application environments, there may be disagreements about the use of a single directory API such as LDAP or JNDI. This model lets developers insulate themselves from this issue by letting the server act as a gateway to a directory service. When agreement is reached, the developers only need to implement a driver for it, and this can be done without having to change or break any of the client code already deployed in the field.

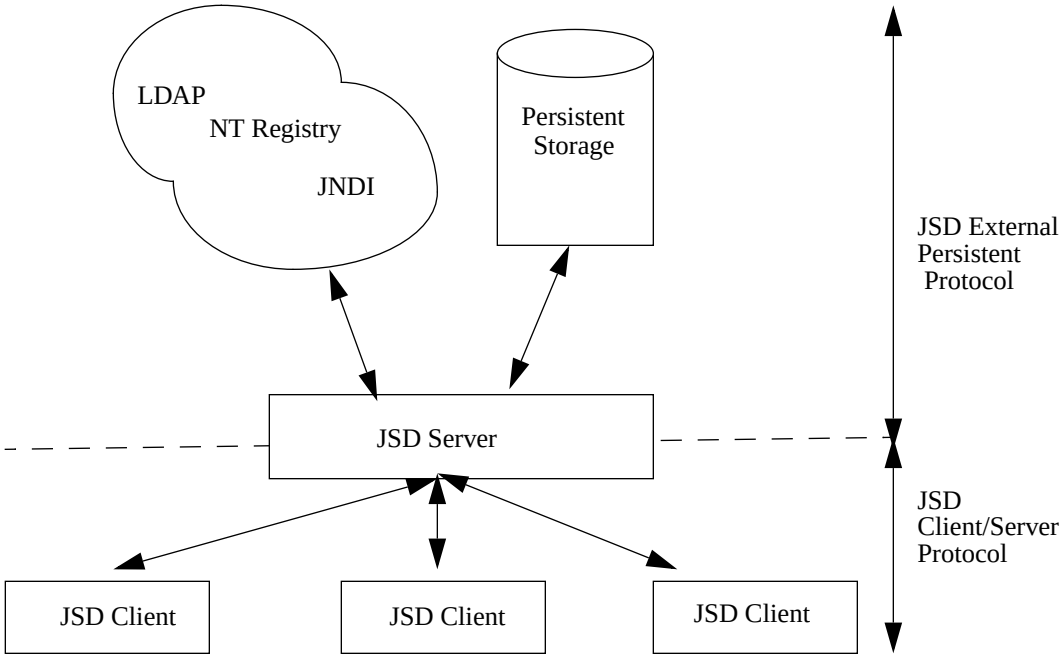


FIGURE 4-4 JSD Client/Server Two-tier Model

The JavaOS for Business software is based on a thin client model. Such a client obtains configuration information, services, applications, and user data access from one or more servers on the network. One server may provide the client the executable and loadable services required to boot the client. Another server may authenticate user login and provide user-specific configuration information to the client. Still other servers may be sources of application and user data.

Server Data Coalescence

Information that is ultimately downloaded to a client comes from a variety of sources within the *config* namespace of one or more servers. Entries and their properties stored in the server JSD are replicated, as appropriate, in the client JSD, as well as any services and applications loaded as a result.

It is the responsibility of the server JSD to impose order on data usage. Typically, more specific information overrides that which is more general. User information has priority over machine information, provided the site security policy is not violated.

Machine-specific information overrides that of the related platform or any profiles. The system administrator decides how one profile overrides another.

User-specific information overrides that derived from the one or more groups to which the user entry refers. Again, security policy may prevent some or all of this from occurring. Likewise, user-derived information may or may not be allowed to override that from the machine hierarchy.

Nonetheless, coalescence of machine and user data occurs on the server, and the result is downloaded into the client *software* namespace. Once on the client, the client schema is the mechanism by which all services and applications used on the client can locate their configuration information.

Client/Server Communication

Figure 4-5 illustrates the high level mechanisms used for implementing persistence within the JSD. Persistent entries, with the aid of a persistent manager class, are pushed and pulled from persistent storage.

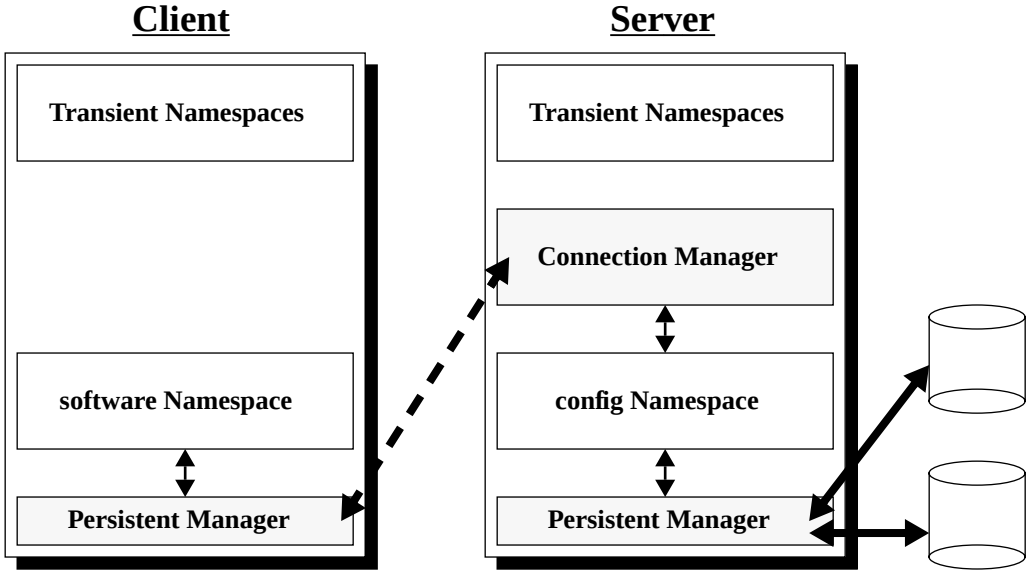


FIGURE 4-5 JSD Client/server organization

In the case of the typical client, the persistent manager communicates its persistent operations to the server via IIOP (Internet Inter-ORB Protocol, a CORBA standard protocol for ORB interoperability layered on TCP/IP to transfer requests between ORBs). A server typically interacts with a mass storage device to achieve persistence.

However, the persistent manager on the server could just as easily communicate with another server using the JSD/IIOP protocol, or it may instead use another mechanism, such as NIS, LDAP, ACAP or some other directory/naming service. A combination of choices can be supported by implementing multiple persistent manager classes.

The persistent manager need only implement the defined persistent manager interface. While the number of potential underlying implementations is unlimited, client and server must use the same protocols.

Client/Server Protocol

A client uses the Client/Server protocol to communicate with the server. The master copy of each persistent entry is stored on the server. When needed, a client issues a cache request to its associated server to cache the entry on the client. The same persistent entry may be cached by multiple clients.

Due to cache-coherency issues related to maintaining multiple copies of a shared entry among different clients and servers, several assumptions have been made to restrict the scope of entry sharing:

- *No Replication*: A persistent entry is stored on only one server. Multiple-server replication is not supported. Clients share entries from the same server.
- *Partitioned JSD Servers*: A client can connect to different servers to get a different set of entries from each server. In the software, each client can connect to at most two servers: a *user* configuration server and a *machine* configuration server.
  - The user configuration server houses user related configuration information. This server contains information necessary for user authentication and setting user and group profiles. When a user signs on, the platform needs information from the user configuration server.
  - The machine configuration houses machine-oriented configuration information typically related to the configuration of the operating system and firmware. When a platform boots, it needs information from the machine configuration server.
  - Both the machine and user configuration information need to be logically combined to provide a single user environment on a particular platform. The separation of user and machine configuration data allows for booting a particular machine from a nearby server, while retrieving user information from a geographically distributed server to support roaming users.
- *Read-Only Sharing*: Persistent entries shared by multiple clients are read-only entries. Each client reads the same entry and writes back updates into its own unique area (user specific area) on the corresponding server. Write-sharing of entries between clients is not supported. A user logged on a client writes updates into their own specific user area on the user configuration server. Two users cannot write to the same entries. However, clients can read the same entries. A

user cannot be simultaneously logged onto two different clients. This restriction is necessary to avoid two clients writing to the same server entry (i.e., user specific area).

- *Server Serialization*: Concurrent client updates are serialized on the server. Only one client request is processed at a time.
- *Unique generation*: Each persistent entry has a unique generation, which is used to resolve conflicts between inconsistent entry values between clients and the server. There is a potential window where an update performed on the server may not have been yet propagated to the client, before the client has updated its own copy of the entry. When an entry is cached by a client, the server registers an event to notify the client if one of the entries it cached has been modified by the server. This mechanism is necessary to keep the server and client entries synchronized. The generation of the client copy before it was locally updated and the generation of the server copy are compared. If the two generations differ, the server entry was modified since the last time the client cached the entry. When an inconsistency is found, the client aborts its update and waits to process the incoming update notification from the server. The server provides a recovery mechanism to guarantee that no deadlocking may occur in case the notification event is lost.
- *Synchronizing transaction writes*: Client transactions involve a single entry or entries belonging to the same branch. A transaction cannot involve updates of unrelated entries. Before the transaction can proceed, write locks are applied to the sub-tree root entry on both the client and server entries.
- *Use of IIOP*: The Client / Server protocol uses an IIOP (Internet Inter-ORB Protocol) based transport. The widely available IIOP protocol was selected to let legacy applications and tools access the server database.

Figure 4-6 illustrates the way client and server communicate.

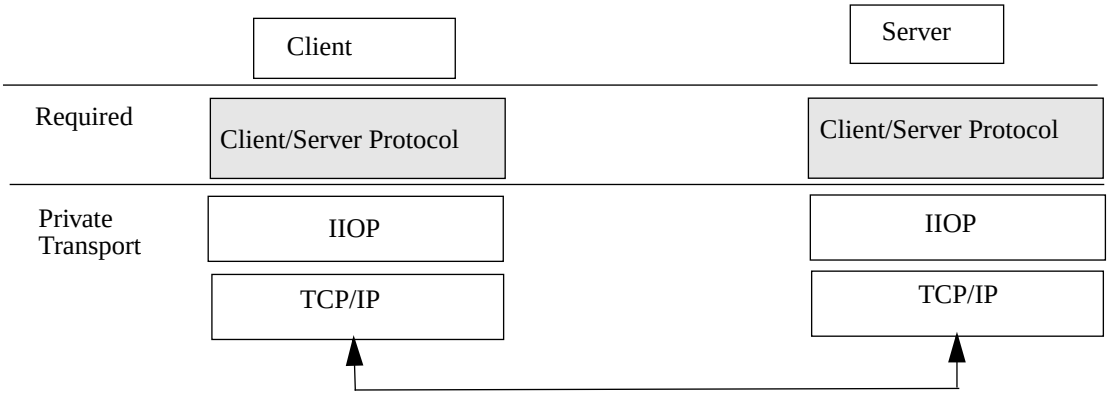


FIGURE 4-6 Client / Server Protocol Stack

IIOP is an implementation choice for the underline transport. Various product families may choose to standardize the protocol stack to insure interoperability between the vendors using a different transport, for example, http over TCP/IP.

Database Schema

The Java System Database is a collection of named database objects. Naming an object is defined as the process of associating a name string with a database entry object. A pathname, typically composed of names delimited by a pathname component-separator character, uniquely identifies an entry in the database.

Entries can represent such things as platforms, files, applications, users, or devices. An entry can also have properties, which are name string and Java object-value pairs. Inserting an entry publishes and advertises its existence to other software components.

For each property, the JSD maintains a reference to the value object. Transient entries do not store the object itself. Persistent entries, on the other hand, are able to convert the value to and from a serialized form so that it can be communicated between client and server. Device properties, for example, list device resources such as memory buffers and interrupt vectors. The JSD provides a standard properties interface used to create, read, or delete properties associated with any entry in the database.

Namespaces

The database is organized as a hierarchy of entries. The hierarchy descends from a single entry known as the superroot entry (/). The superroot is created and initialized during the Java start-up procedure.

Every entry in the hierarchy *may* act as both a single entry and the root of a tree of descendant entries. Each entry in the tree has a single parent and may have an unlimited number of child entries. All entries contain links to their parent entry and children.

The database hierarchy is further subdivided into namespaces. A namespace is a specially designated tree of entries that name objects of like kind, such as software or devices. Namespaces are always direct descendants of the superroot. A number of standard namespaces are created during the platform start-up procedure.

Namespace names, their schema, and how they are populated, along with the underlying persistence mechanisms, if any, are not dictated by the core JSD itself.

Namespace Manager

Each namespace is managed by a default namespace manager. The namespace manager controls how the entries are actually stored and accessed within the namespace. A namespace manager implements a standard interface that exports the security, storage, and ownership attributes of any entry in the namespace An entry inherits its parent's manager when the entry is inserted into the database.

Not all namespaces need be completely populated with all known objects of a particular kind. The namespace manager controls the granularity of the object population.

Standard Database Namespaces

The Java platform initialization sequence creates six standard namespaces that are available in all implementations of the platform. Other dynamically constructed namespaces may be added to the database after initialization is completed.

Each standard namespace has a namespace manager. Additional namespaces also require namespace managers, which must be provided by the creator of the namespace. The six standard namespaces provided in the JavaOS for Business software, as illustrated in Figure 4-7, are *Software* and *Config*, which are persistent, and *Temp*, *Device*, *Interface*, and *Alias*, which are transient

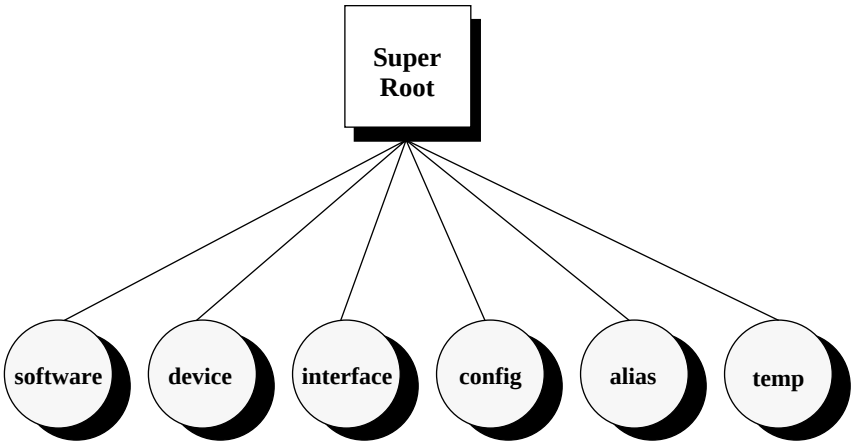


FIGURE 4-7 Standard Top-Level Namespaces

It is important to note that the server and the client have different subsets of the standard namespaces. Only the persistent *config* namespace exists on the server. All other namespaces exist only on the client.

Namespaces are summarized in the following table:

TABLE 4-1 Namespace Summary

Namespace	Population mechanism	Description
software	Populated from server during boot and user login	*application*, *system*, *service* and *public* subdivisions hold persistent data on client
device	Populated by *Tree Populator* Interface	Subdivisions are bus / device-specific
interface	Populated by *device* manager	Holds driver information with cross-references to *device* and *alias* namespaces
config	Server-side JSD from persistent storage	Used on *server side only* to store persistent information
alias	System administrator	Cross-reference to *interface* namespace
temp	Any subsystem or application	Temporary storage

Software Namespace

The *software* namespace contains the list of installed and / or available system services such as device drivers, applications, and user configuration information. This information appears as a hierarchy downloaded from one or more server machines during boot and user login.

The *software* namespace is persistent, in that the server provides permanent storage for all entries in this name space. The organization of the *software* namespace defines much of the publicly accessible client side JSD, the top level of which is illustrated in Figure 4-8.

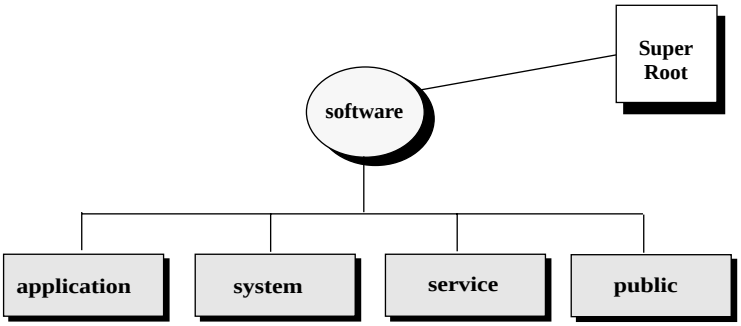


FIGURE 4-8 Sub-schema of the Software Namespace

Under each *software* category (i.e., application, system, etc.) are sub-entries that use the unique JavaOS for Business software naming scheme for entry names. Company unique names such as *com.ibm*, *com.sun*, *com.lotus*, *com.nc*, etc., distinguish company specific information. Entry names and organization below the company unique entry are company specific.

Beneath each category, the sub-entry *java* represents entries common to the version of the JDK implementation. These are not company specific.

Application

A sample *application* sub-entry hierarchy is illustrated in Figure 4-9.

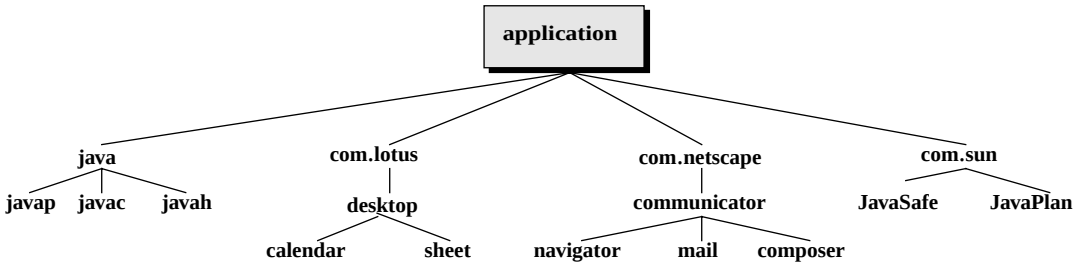


FIGURE 4-9 Application Hierarchy

Properties can be assigned to each entry to denote configuration information for a specific application. The application's class path serves well to organize entries. Additional entries could be created to represent preferences for different versions of the same application. The organization below the individual company level is vendor-specific.

System

The *system* hierarchy, an example of which appears below in Figure 4-10, contains information pertinent to runtime operations, including virtual machine and NC operating system- specific information. Loadable components, such as device drivers, maintain their configuration information in the *service* hierarchy.

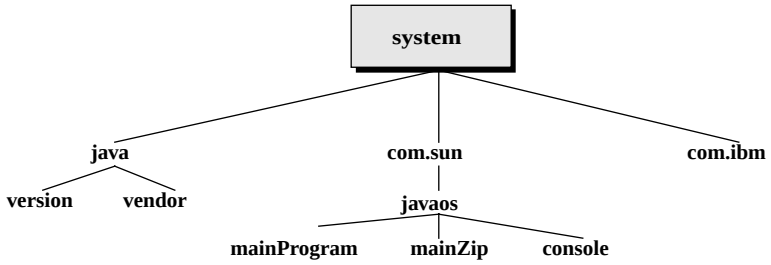


FIGURE 4-10 System Hierarchy

Service

Services loaded by the system maintain their configuration information in this hierarchy. A service is a system component such as a device driver, networking stack, file system, etc. Figure 4-11 illustrates one possible *service* hierarchy.

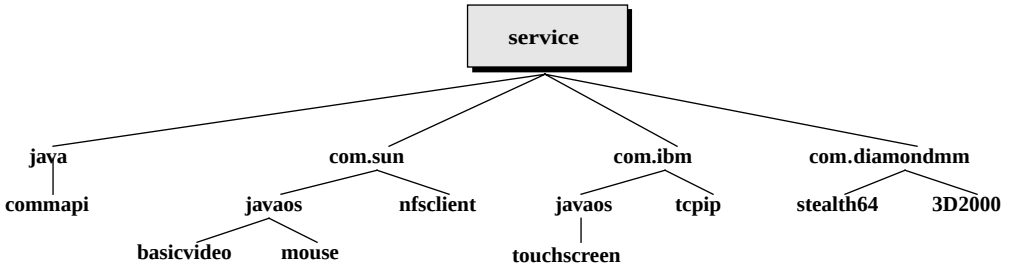


FIGURE 4-11 Service Hierarchy

Public

The *public* hierarchy stores data that is typically global, rather than have each service or application define values for that data.

However, any service or application may choose to override the public setting with its own. Figure 4-12 shows a hierarchy which includes such common information as proxies and mail server information.

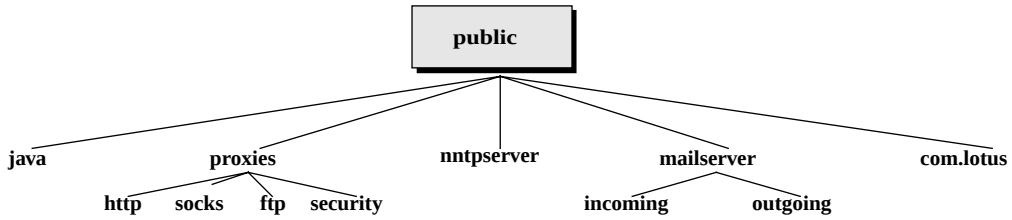


FIGURE 4-12 Public Hierarchy

This hierarchy is also available for an individual company to make information public to the rest of the system, without requiring that it be stored in vendor-specific portions of the service and application hierarchies.

Config Namespace

The *config* namespace maintains client, user, and application configuration information. This namespace is used *only* on the server-side JSD implementation. The *config* namespace contains two kinds of information: machine and user. Relevant information in both are downloaded into the *software* namespace on the client, as shown in Figure 4-13.

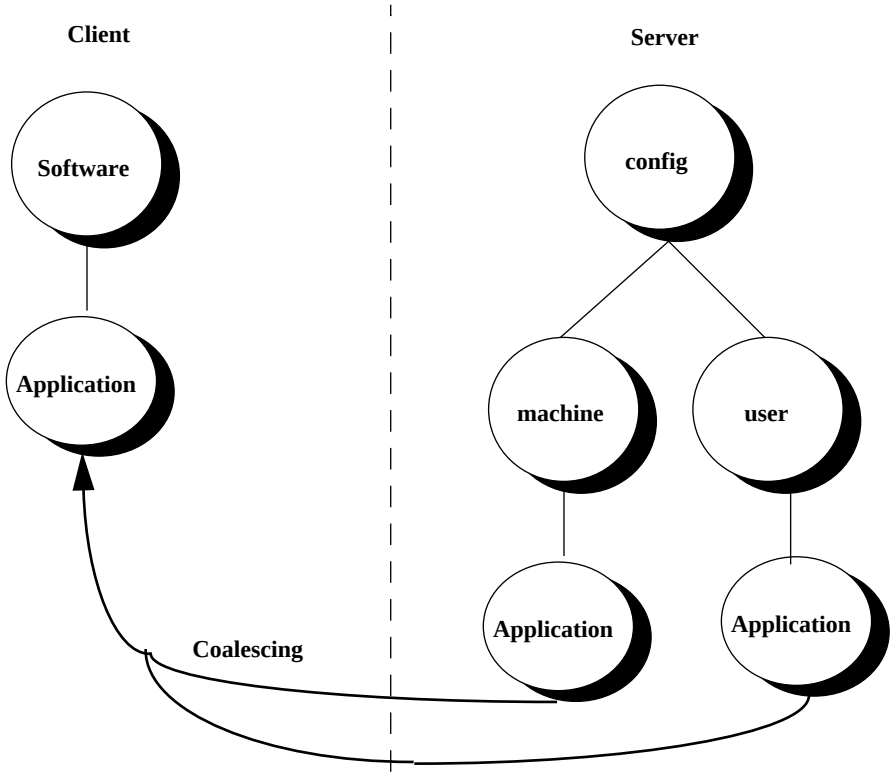


FIGURE 4-13 Client / Server Schema Relationship

Machine

All client machines must identify themselves to the server via a machine unique identifier string. This is typically the MAC (Media Access Control) address (ethernet, token ring) and a hardware type. Given the identifier string, the server can locate information specific to the client in */config/machine/identifiers/mach_unique_id*.

A machine identifier entry typically refers to a platform entry that allows sharing of common information and easier administration. For example, information common to a given family of clients can be found in the *platform* sub-entry, e.g.,*/config/machine/platform/JDM1*.

If the system administrator decides that a specific client is to provide additional information, or to override that in the platform, this information is maintained in the machine-specific entry.

A machine identifier may refer to one or more *profiles*. A profile allows for additional machine groupings that share common properties. For example, in a retail situation all of the clients in the sportswear department may refer to a profile that is different from that referred to by the clients in the jewelry department. The profile is not dictated by the machine architecture (as is the platform), but by other relationships determined by the system administrator.

Together, the machine unique entry, its platform, and profiles determine what services, applications etc., are downloaded to the client when it boots, but before a user logs in. Figure 4-14 illustrates the *Machine* hierarchy.

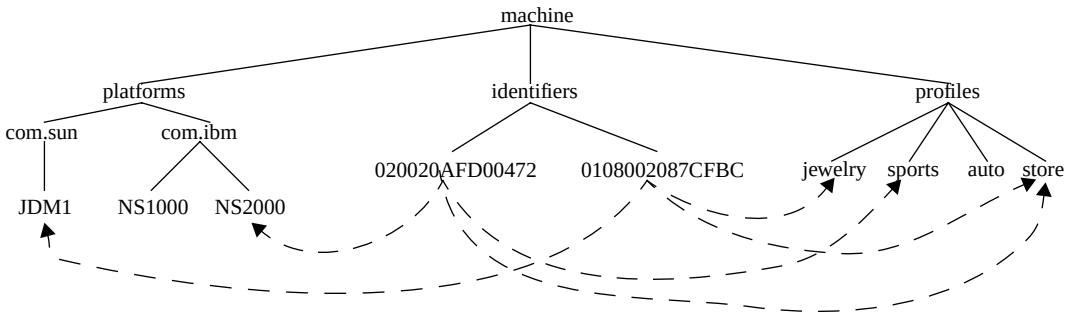


FIGURE 4-14 Machine Hierarchy

User

The user information is similar to those provided for machines. A user is identified by a login identifier such as *chris*. The user entry is the only place where users may customize their attributes without system administrator intervention. Individual users can specify the additional services be loaded upon login, or that certain attribute values override those defined in a group entry. If permitted, user attributes may even override those obtained from the machine entry hierarchy.

A user entry may refer to zero or more group entries. A group entry provides information common to one or more users. Groups are set up and maintained by the system administrator.

The *user* hierarchy is illustrated in Figure 4-15.

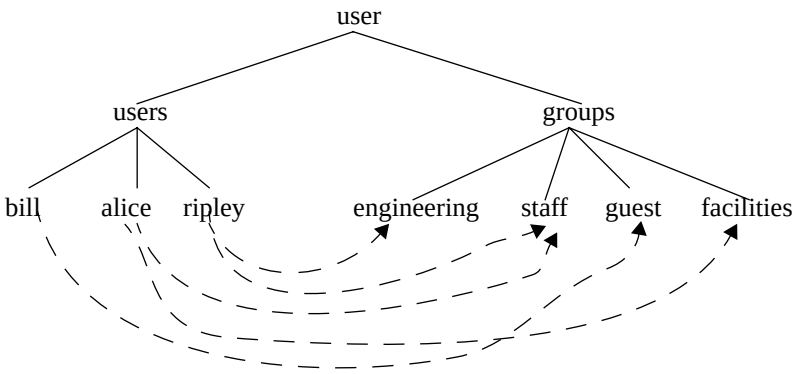


FIGURE 4-15 User Hierarchy

Some server entries are read-only entries, others can be written to by a client. Applications read data from client-cached entries. Writes to persistent entries are pushed to the user entry space on the server. For example, a client application can only read entries in the machine specific area (*/config/machine/platform*) and write entries to the user specific area (*/config/user/bill*). The general framework of the server schema is for a client to be only able to write to the *user/users* area of the user currently logged on. However, the client can read any other areas outside of some specific permission restrictions. A user may not be allowed to read another user's information.

The server schema defines an implicit overwriting hierarchy where information found in the user area supersedes information found in the machine area. The final client entry is obtained by *coalescing* or merging entries in the following ranking order:

1. identifier entries
2. platform entries
3. machine profile entries
4. group entries
5. user entries

User entries override all other information, subject to the security policy implemented.

Device Namespace

The *device* namespace contains the set of devices discovered on the local platform. The structure of the *device* namespace reflects some or all of the I/O bus and device hierarchy present on the platform. The physical connectivity of buses and devices are represented as a tree of entries where a bus is a parent and leaf entries are devices.

Device Entries

The *device* namespace manager oversees the *device* namespace contents. As physical or logical devices are discovered, a *device* name entry is created to represent the resource to drivers and applications. *Device* name entries in this namespace can also represent virtual devices like RAM disks.

The hierarchy in the *device* namespace directly reflects the geographical location of the represented devices, including the bus used to access the device.

In general, leaf entries (those entries without children) are devices and parent entries are buses. The bus entry is always a parent or grandparent to a device entry. Bus entries may also be children of other bus entries. For example, a PCI bus bridge may lead to a S-Bus containing an S-Bus device.

This scenario is illustrated in the central panel of Figure 4-16, where the PCI bus entry is the parent of an S-Bus device entry, which in turn is the parent of an S-Bus device.

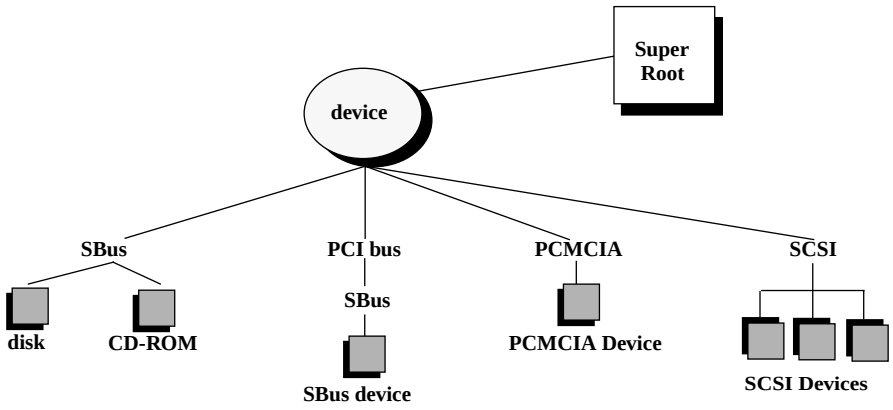


FIGURE 4-16 Bus-to-Device Topology

The *device* namespace is constructed during the Java start-up process. The *device* namespace manager (also called the *device* manager) uses the services of the Tree Populator interface to build the *device* namespace one entry at a time.

The Tree Populator gives the *device* manager property information about each platform device that aids in finding drivers.

Interface Namespace

The *interface* namespace is a cross-reference of objects by type in other namespaces (such as *device* or *software*).

For example, device drivers compete to manage entries in the *device* namespace. Once the driver claims the device, a cross-reference *interface* entry is inserted under the device entry in the *interface* namespace. For example, as serial device drivers claim serial devices (in the *device* namespace), a cross-reference `SerialDevice` entry is added to the *interface* namespace. Figure 4-17 illustrates the relationship for a serial mouse.

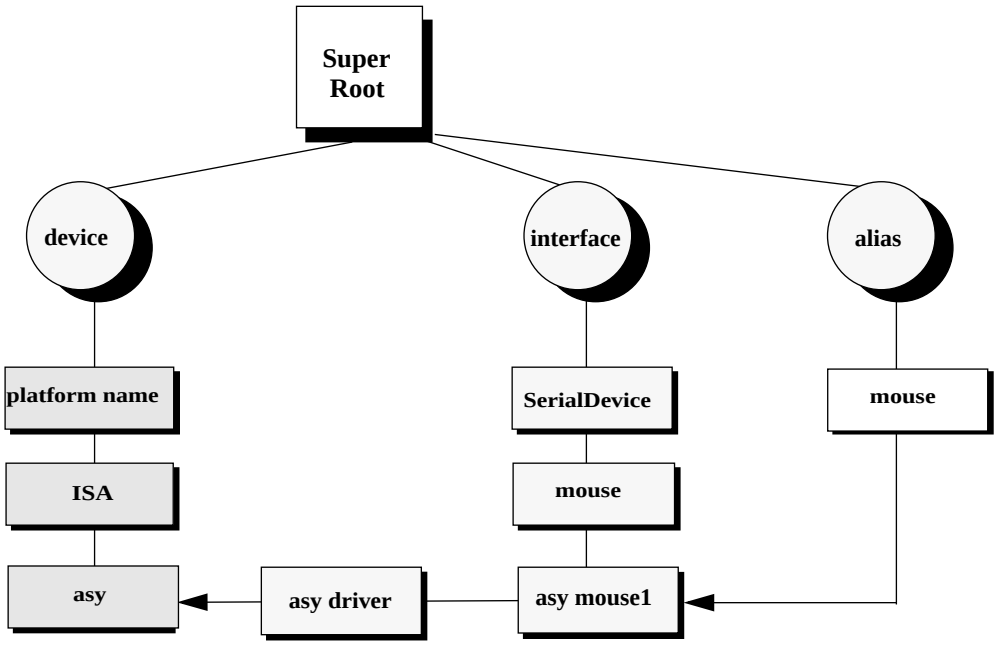


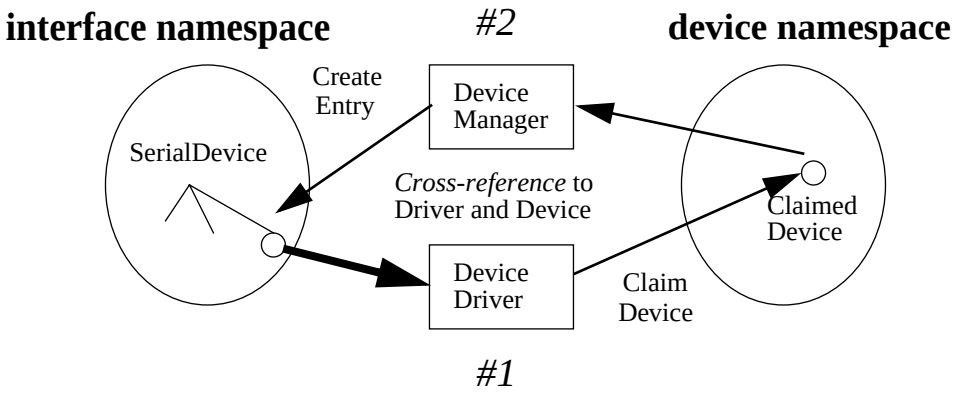
FIGURE 4-17 Interface Topology

Java applications, applets, and JavaBeans search for devices by interface type in the *interface* namespace, and are thus insulated from the physical connectivity of the platform. Applications may use the more simplified *Alias* namespace to locate common devices, such as */alias/mouse* illustrated above. (For more on device drivers, see Chapter 2, The JavaOS Device Interface (JDI).)

Interface Entries

The *interface* namespace contains the set of device drivers that implement a standard Java language programming interface. As a driver is assigned a device, the *device manager* notes which interfaces are implemented by the driver. These interfaces are then automatically recorded in the *interface* namespace.

The Java Service Loader (JSL) populates the *interface* namespace with driver cross-reference entries to appropriate devices. This cross-reference assignment process is illustrated below, where (#1) a device driver implements an interface and (#2) the device manager assigns a device to the driver in the *device* namespace.




```
class DeviceDriver implements SerialDevice {
.....
}
```

FIGURE 4-18 Interface Namespace Assignment

The contents of the *interface* namespace change as new devices are discovered or existing devices are removed. However, there is no one-to-one correspondence between interface entries and device entries, because a single device driver may support more than one programming interface (for example, *SerialDevice* and *PowerManagement*, introduced in Chapter 9, Class Files.

The *interface* namespace topology is rather flat, containing a parent entry for each kind of Java interface implemented by a device driver. Each parent may have zero or more children representing the devices that implement the parent interface.

For example, serial drivers manage serial ports and implement the SerialDevice interface. The JSL creates an entry in the *interface* namespace for each serial port actively managed by a serial driver. The SerialDevice entry is created as a child of the SerialDevice interface entry.

Alias Namespace

The entries in the *alias* namespace refer to entries in the *interface* namespace and provide a simple, logical naming scheme for some system devices. Aliases can be explicitly defined by the system administrator via the business cards used by the JSL. Alias entries themselves are transient.

An example of a common alias in this namespace is:

/alias/system/DefaultPrinter

which could refer to the entry:

/interface/device/printer/hpInkJet#1

All aliases (type `SystemAliasEntry`) are confined to the *alias* namespace. No other type of entry may be inserted into the *alias* namespace.

Temp Namespace

The *temp* namespace is available as temporary storage. While events are generated to indicate changes to entries in *temp*, the entries themselves are not persistent. Applications can take advantage of the JSD feature set and use this temporary storage facility.

JSD Instantiation

An instance of the `SystemDatabase` class is the JSD. Only one instance of this class is allowed to exist within a given Java Virtual Machine. The class constructor is defined as:

```
public SystemDatabase(Manager[] managers) throws SystemDatabaseException;
```

The special superroot entry is created (of type `SystemEntry`) and assigned the default manager class `SuperRootManager`. This entry is put into the published state and its parent reference is set to itself. The superroot entry is the only entry in the JSD that can refer to itself as its parent. Code within the JSD takes advantage of this fact. The `SystemDatabase` constructor also establishes the Event System's exclusive event producer used to produce JSD related events.

Zero or more manager class references may be provided to the `SystemDatabase` constructor. It is the responsibility of the managers to establish the initial namespaces, and possibly entire hierarchies (i.e. tree population). The code instantiating the JSD first constructs the desired initial managers, and then their references are provided to the `SystemDatabase` constructor, which will invoke the *init* method of each of the managers.

This two phase process allows for a manager's constructor to be arbitrarily complex. When the `init` method is invoked, it is provided the reference to the JSD superroot entry:

```
public Entry init(Entry parent) throws SystemDatabaseException;
```

The manager returns the namespace root entry it created. It must also insert the namespace root under the superroot, and override the entry's manager reference if desired.

The `SystemDatabase` methods `getSystemDatabase` and `getSuperRootEntry` return a `Tree` representing the JSD or just the superroot entry reference respectively.

Entry Anatomy

The JSD is comprised of a hierarchy of entries, each of which is a unit of information identified by a pathname. An entry may have one or more properties associated with it. Properties provide further description of the entry.

The `Entry` and `TransactionFactory` interfaces define the public API for operating on entries in the JSD. All methods defined in these interfaces are implemented via final public methods within the `BaseEntry` abstract class. The `BaseEntry` class supports the following entry attributes, which are used to maintain the database hierarchy, management scheme, and the properties associated with each entry:

- Name
- State
- Parent
- Children
- Lock
- Manager
- Generation

- Properties

The `SystemEntry` concrete class is a subclass of `BaseEntry`. `SystemEntry` is the standard transient entry type used within the JSD. This class provides the actual implementation for manipulating properties associated with the entry.

The illustration below illustrates the interface and class hierarchy used to implement entries within the JSD. Pay special note to the key position played by the `BaseEntry` abstract class.

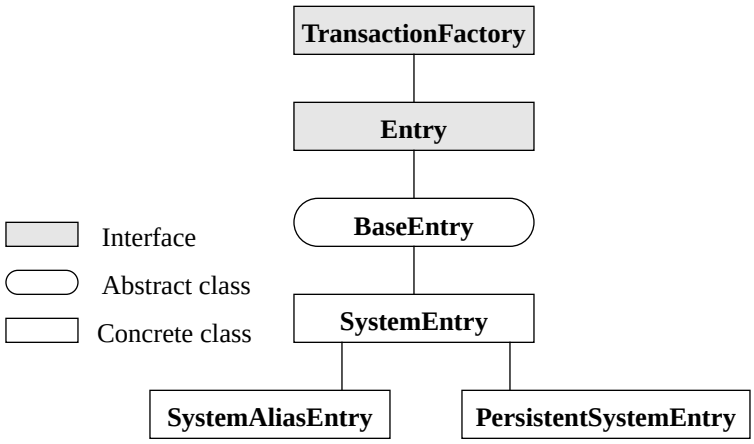


FIGURE 4-19 JSD Entry Class Hierarchy

BaseEntry Details

By providing the public API implementation, the `BaseEntry` class allows important tasks such as transaction creation (the factory) and transaction and locking usage to be centralized.

The JSD superroot is an instance of `SystemEntry`. The public API implementation of the insert methods (implemented in `BaseEntry`) allow only the insertion of children that are derived from the `BaseEntry` class. Thus, all entries published in the JSD are of a known heritage. Other classes directly implementing the `Entry` interface cannot be inserted into the JSD. While this does limit flexibility to some degree, knowing that all entries are well behaved is very important to the robustness and security of the database.

The `BaseEntry` abstract class implements the entry public API (defined by the `Entry` and `TransactionFactory` interfaces), but it also defines two additional APIs as indicated in Figure 4-20.

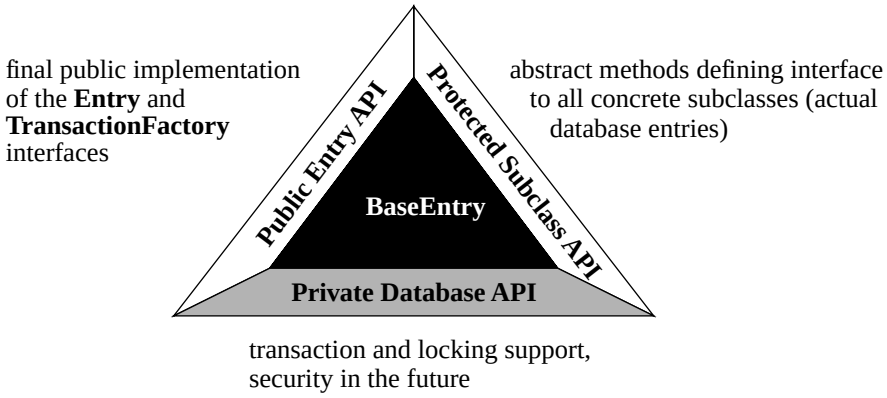


FIGURE 4-20 BaseEntry APIs

Protected Subclass API

A number of protected abstract methods are defined that all concrete subclasses (including `SystemEntry`) must implement. Methods are defined to validate the insertion, removal and disconnection of children and those for manipulating properties. There are additional methods used to indicate persistence, verifying entry names, obtaining the pathname component separator defined for the entry class as well as for locating pathname components of children. The following diagram illustrates the relationship between `BaseEntry` and subclasses:

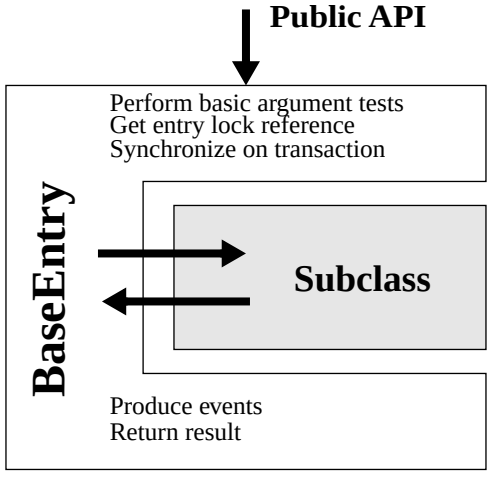


FIGURE 4-21 Sub-class API Relationship

The purpose of the protected subclass API is to allow developers to implement custom entry classes without the burden of understanding, and implementing, the core JSD transaction, locking, and security methodologies. More importantly, entry classes are forced to participate in a well known manner since they are subclasses of `BaseEntry`.

Note: This API is *less public* than the JSD core public API. It is meant for developers of database entry classes rather than users of the JSD itself. It is not recommended at this time to develop to this API as it is subject to significant change over the coming months.

Package Private Database API

Within the database package, all entries must provide methods to obtain and return their lock objects and provide methods for committing and aborting operations. These methods are implemented by `BaseEntry`, and are invoked by other components of the JSD.

See Chapter 9, Class Files, for more details on the methods comprising the sub-class API.

BaseEntry Attributes

Name

The entry name is a `java.lang.String` object, and can contain any URL addressable characters. There are no other restrictions except that the name cannot be a null reference or empty string, and the name must be unique among all of the siblings of the parent under which the entry resides.

The *SystemEntry* class defines an additional restriction which is that the pathname component separator character, the forward slash (/), cannot be used in the name string.

State

An entry has three distinct states that it may enter during its lifetime:

- drafted
- published
- deleted

Drafted State

Entries are initially constructed in a drafted state that exists outside the bounds of the database, but inside Java's object heap. The creator of an entry constructs the entry with a name using a snippet of code something like:

```
SystemEntry sample = new SystemEntry("SampleName");
```

Changing a drafted entry does not produce an event. Therefore, the creator of the entry can add, change, or delete properties and to achieve the desired property set, and then later publish the entry for the rest of the database to see. Entire hierarchies of entries may be built off-line and then inserted as a subtree in a single operation.

An entry, or hierarchy of published entries, may be disconnected from the JSD and be returned to the drafted state, where they are considered once again to be off-line. However, since references may be held to disconnected entries, they are not truly off-line. While events are not generated to reflect changes to such entries, the generation is updated even for drafted entries.

Published State

An entry enters the published state when it is inserted into the database under a published parent. Once the entry is in the database, interested applications and/or system services can locate the entry using its name and/or its properties as search criteria.

When an entry is inserted, the JSD generates an insertion event. Listeners interested in investigating new entries are notified in this manner. Also, property events are generated by the JSD when necessary for all published entries.

Deleted State

When an entry is removed from the JSD, it enters the deleted state, regardless of whether it was in either the off-line drafted state or the on-line published state. All method calls fail for deleted entries; an `EntryDeletedException` is thrown. A deleted entry is essentially waiting for garbage collection. However, since there may still be references to deleted entries, whenever a method call is made on a deleted entry it will throw an `EntryDeletedException`. A deleted entry cannot enter any other state.

Figure 4-22 illustrates the state transitions an entry may undergo during its lifetime.

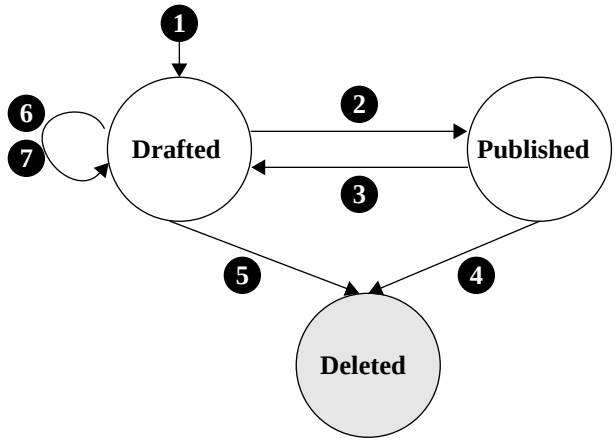


FIGURE 4-22 Entry State Transitions

The numbered transitions are defined as:

1. Entry is constructed.
2. Entry is inserted under a published parent entry. An *EntryInsertEvent* is generated.
3. Published child entry is disconnected from (published) parent. An *EntryDisconnectEvent* is generated.
4. Published entry is removed, which generates an `EntryRemoveEvent`.
5. Drafted entry is removed.
6. Drafted entry is inserted under a drafted parent entry.
7. Drafted child entry is disconnected from (drafted) parent.

Parent Entry

While in the published state, the parent entry is always non-null and refers to a published entry. The superroot refers to itself, as it has no parent. The parent of a drafted entry may be null, indicating that the entry is the subtree root of a hierarchy of entries, the degenerate case being a single entry. An entry has at most one parent.

Child Entries

An entry maintains references to all its children. The child references are available through enumeration (see the section on Database Navigation, later in this chapter).

Lock

A reader-writer lock mechanism (`SystemDatabaseLock`) lets an entry be inspected or modified without interference. The Java monitor mechanism (synchronized keyword) provides for an object to be exclusively locked within the scope of the synchronized keyword (code block or entire method). However, the database entry lock allows for multiple method calls to be made while the entry is locked. It also allows for multiple concurrent readers.

Locks can be acquired for either shared or exclusive access. The methods defined in `TransactionFactory` (implemented by `BaseEntry`) determine in which mode a lock is acquired on an entry. Multiple shared transactions may access the same lock simultaneously. Once an exclusive transaction acquires a lock, all other transactions are blocked until the lock is released. See the Transactions section later in this chapter for more details.

Manager

A loadable component of software called a *manager* manages each entry. Each namespace provides a manager class. An entry inherits the manager of its parent when it is inserted into the database.

The manager sets policy for the entry. It may perform security checks or make other decisions that affect the behavior of an entry. In addition, the manager is the mechanism by which persistence is provided for the entry.

Generation

The entry generation is a value that is updated whenever the entry is changed, while either published or drafted. The generation provides a polling mechanism for simple coherency schemes not requiring the overhead of event processing.

A change to an entry is defined as either insertion, disconnection, or removal, or any property additions, changes, or deletions. Hierarchy changes result in the generation of both the parent and the child being updated. Property related changes update the generation of only the single entry.

The getGeneration method returns an instance of Generation which represents the curent generation of the entry. The Generation class defines the equals method usd to compare one Generation with another. If equals returns true, then the entry has not been updated since the time the older Generation reference was obtained.

Properties

Associated with each entry are zero or more properties. A property is comprised of a name (java.lang.String) and a value (java.lang.Object). Methods are provided in the *Entry* interface to add, change, delete, and retrieve properties. Property names may also be enumerated to discover all those present.

No further structure is applied to properties. The name and actual type of the value are strictly up to the user of the property. The JSD supports an implicit *Boolean* property. If a property is added and no value is provided, then the property is considered boolean, where its presence denotes a value of true and its absence indicates a value of false.

While the BaseEntry class does not itself implement properties, it does implement the public entry API methods used to access and manipulate them. These methods perform the proper transaction/locking protocol and then invoke the subclass API to access properties.

Methods may add, remove and change properties. A change occurs when an existing property is assigned a value. If the value object itself is changed in some manner (e.g., bytes changed in a byte array) rather than the property value (the reference), the JSD is unable to detect that a change has occurred. It is the responsibility of the entity making such a change to notify others by assigning the same property name and value to the entry again. This results in updating the generation and producing an event.

Invoking the getPropertyCount method returns the number of properties associated with the entry. The getPropertyNames method returns a java.util.Enumeration whereby all property names associated with the entry can be enumerated. Given a property name, its value is obtained via the getPropertyValue method.

The addProperty method adds a new property or modifies an existing one. If the name of the property to be added does not already exist, then one is created and the provided value is associated with the new property name.

If the entry is in the published state, then an EntryPropertyInsertEvent is produced for interested consumers. The addProperty method returns a null Object reference to indicate a new property.

If the property name already exists for the entry, then the new value is associated with it and for published entries an EntryPropertyValueChangeEvent is produced. In this case addProperty returns the reference of the previous value object.

To remove a property from an entry, use the `removeProperty` method. When a property is removed, an `EntryPropertyRemoveEvent` is produced for published entries and `removeProperty` returns the reference to the value of the removed property.

Entry Operations

Three fundamental operations are defined for the JSD entries: insertion, disconnection and removal.

Insertion

Entries are constructed in the drafted state. A drafted entry can be inserted under either a drafted parent entry or a published parent entry. In the former case, the child remains drafted; in the latter, it changes to the published state. When published, the child also inherits its parent's manager. This procedure applies equally to a subtree of entries (when inserted under a published parent entry) as it does to a single entry. An `EntryInsertEvent` is produced for each entry in the subtree, and the entry inherits its parent's manager.

Insertion is performed by invoking the `insert` method on the parent entry and providing the reference to the child. The parent determines the suitability of the child, and may refuse to insert it. For example, if the parent entry is an alias

(`SystemAliasEntry`), the insertion will fail if the child is not of type `SystemAliasEntry`. Likewise, an entry may only allow certain types of children, or adhere to some other form of family planning. See Figure 4-23.

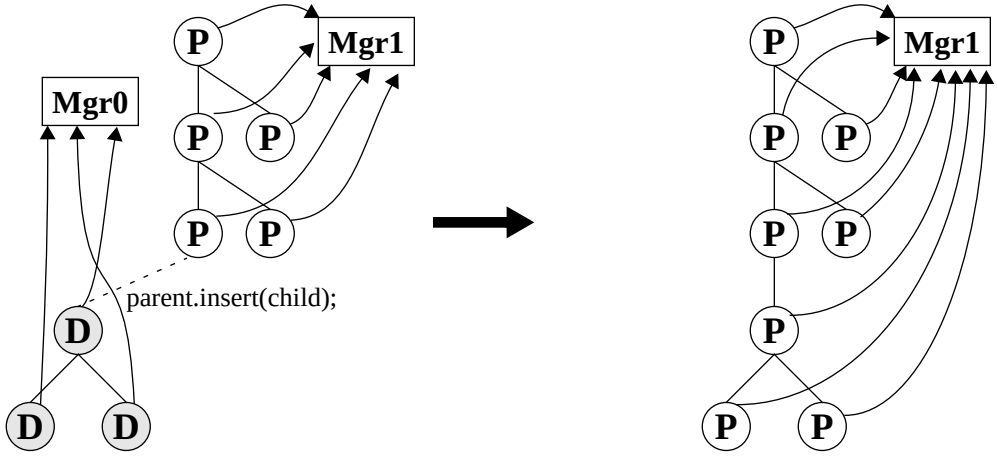


FIGURE 4-23 Subtree Insertion

The insert methods, as implemented by `BaseEntry`, verify that the child to be inserted is an instance of `BaseEntry`; that is, it is a concrete subclass of `BaseEntry`. This prevents the insertion of rogue `Entry` interface implementations.

If the child entry to be inserted already has a parent, it is automatically first disconnected from its current parent, and then inserted under its new parent. No exception is thrown.

Disconnection

An entry (or subtree) can be disconnected from either a drafted or published parent. The parent's `disconnect` method is called giving the reference of the child to disconnect. When a child is disconnected, all descendants of that child are disconnected as well. When disconnecting published entries only, an `EntryDisconnectEvent` is produced for each.

The parent-child relationship is severed at the disconnection point only. All descendants of the child retain their familial relationships (the subtree) even though they may be disconnected. All entries in the subtree change to the drafted state but their manager references are not changed. See Figure 4-24.

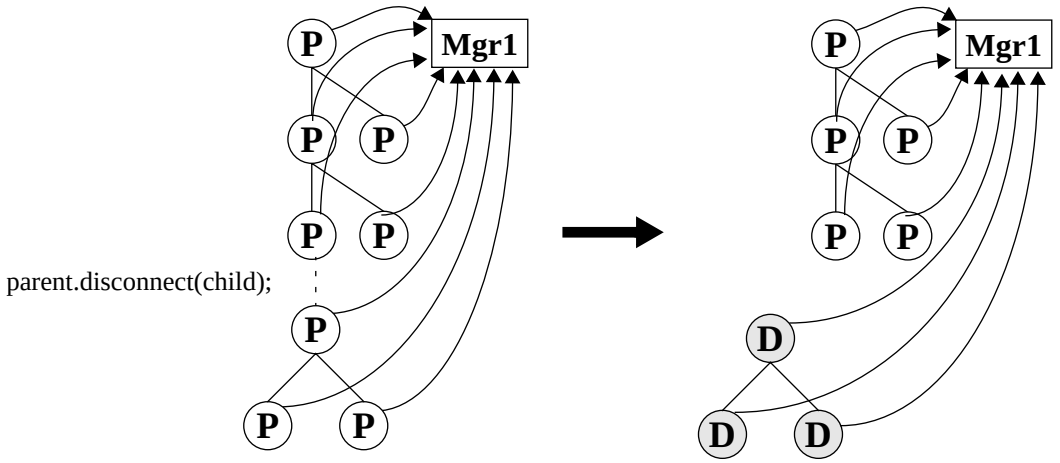


FIGURE 4-24 Subtree Disconnection

Disconnection is not supported by the persistence implementation provided by `PersistentSystemEntry` and `PersistentManager`.

Manager references are left unchanged since entries disconnected from a published parent are still possibly accessible if references to them are held. Therefore, the manager reference remains to manage any method calls made on disconnected entries.

A subtree of drafted entries can be just as easily disconnected from a drafted parent, in which case the procedure is identical, but no events are produced.

Removal

Removing an entry (or subtree) is a two step process, as shown in Figure 4-25. Since all operations must be able to be rolled back in the event of failure, the first step in entry removal is to disconnect it. If disconnection succeeds, then the actual removal occurs when the transaction is committed.

In order to encourage garbage collection, all references held by the removed entry are set to null. This includes all familial references, manager, properties, lock, etc., as well as the aliased entry reference in the case of an alias.

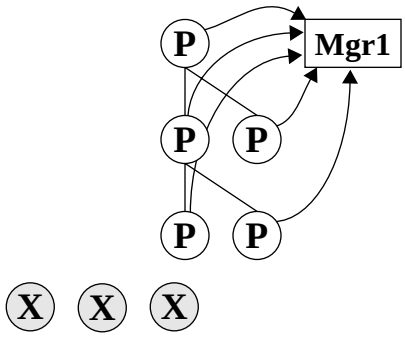


FIGURE 4-25 Subtree Removal Final Step

Just as with disconnected entries, references still may be held to deleted entries if they were once published in the JSD. Any method call on a deleted entry results in an `EntryDeletedException` being thrown, indicating to the reference holder that it should be released and the deleted entry eventually picked up by the garbage collector.

Events

The system database generates events to indicate changes to the database. All database events are descendants of the `SystemDatabaseEvent` base class. Thus, the class of the event object indicates the type of event received. The event object `getEntry` method returns a reference to the affected database entry.

The Event System (`javaos.javax.system.events`) matches event producers and consumers and delivers the event objects as required. The Event System provides an event filter string, which is used by the JSD to better target consumers with event objects. A JSD event contains a string representing the pathname to the entry to which the event pertains. The filter can then be applied to further refine delivery of an event.

Consumers interested in JSD events register directly with the Event System. As of this release of JavaOS for Business, there is no JSD API for event registration. The JSD registers with the Event System as an exclusive producer of all of the JSD event classes. Thus, JSD events may not be produced by any other source.

With event filtering it is possible, for example, for an event consumer to only be interested in events pertaining to entries in the */software* namespace. This listener can register the filter string */software* and only events containing filter strings beginning with */software* will be delivered to the consumer. This filtering is done by the event producer in order to eliminate unnecessary context switches to consumers.

The Event System filtering mechanism is general purpose in nature. It is used specifically by the JSD to allow for specific types of events so that consumers may elect to receive only the event class types of the scope in which they are interested. The filter string only serves as a prefix (`java.lang.String.startsWith`). General regular expressions and the like are not currently supported. Thus, it is not possible to set the event scope so that events in, say, */temp* and */config* are considered interesting. A scope of null or / results in all events of the specified class being delivered. For more details, see Chapter 5, JavaOS for Business Event System.

There are two major JSD event categories:

- Those that pertain to the database hierarchy (such as insertion, disconnection and removal of entries)
- Those that indicate changes to the properties of an entry (such as adding, removing, or changing a property).

New event categories can be introduced in the future by subclassing `SystemDatabaseEvent` or one of its subclasses.

The Event System subsystem allows for event consumers to take advantage of Java class hierarchies. By listening for `EntryPropertyEvent`, the listener will receive events of type `EntryPropertyEvent` and all of its subclasses. Thus, all events pertaining to property changes (though not entry insertion, disconnection and removal) are received, subject to any specified scoping.

Event Class Hierarchy

The JSD event class hierarchy is illustrated in Figure 4-26.

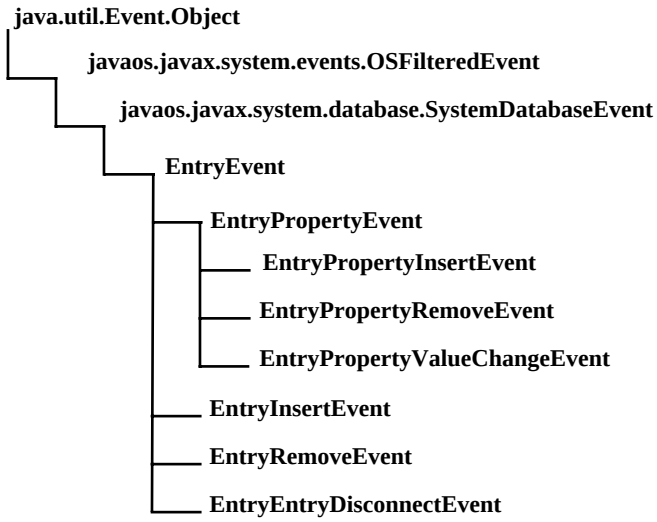


FIGURE 4-26 Entry Hierarchy

Listening for `SystemDatabaseEvent` results in the reception of all database events. Event class types can be distinguished via the Java `instanceof` operator. `EntryEvent` and all its subclasses contain a reference to the affected entry. `EntryPropertyEvent` and its subclasses can be queried for the name of the affected property. The previous property value (object reference) is available from `EntryPropertyRemoveEvent` and `EntryPropertyValueChangeEvent`.

Managers that register as event listeners can take appropriate action upon receipt of an event. For example, the PCMCIA manager can listen to insertion and removal events and then act appropriately when a PCMCIA card is either inserted or removed by the user, and load or unload the related services and add or remove the interface entries pertaining to the card.

The event response by one level of listener results in the generation of events that causes another listener to take action. The insertion of a PCMCIA modem card, for example, causes the PCMCIA manager to load the needed services (drivers, etc.) for the card, and then create an entry in the *interface* namespace. This in turn generates another event for which the *interface* namespace manager attempts to locate a higher level service, such as a dialer applet.

Exceptions

Exception classes potentially thrown by the core JSD are listed in the figure below:

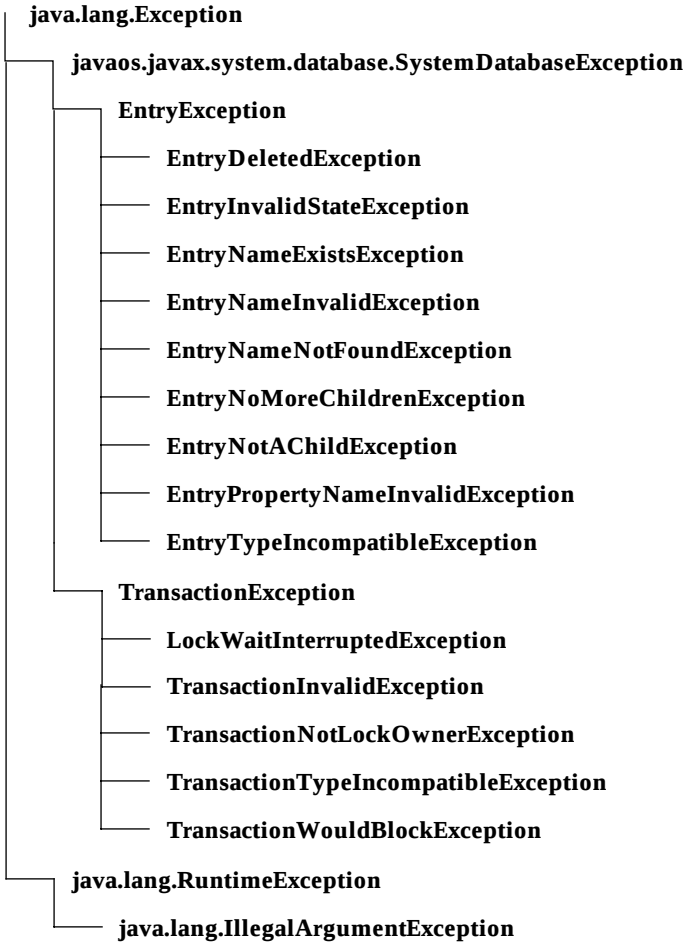


FIGURE 4-27 Exception Classes

The `SystemDatabaseException` class is the top level of the JSD specific exception hierarchy, all subclasses of which are checked exceptions. `IllegalArgumentException` (unchecked) can be thrown when arguments provided to JSD methods and constructors are completely bogus, such as null entry or transaction references, or if an entry reference pertains to an entry that is not a subclass of `BaseEntry`.

Exceptions are grouped in two broad categories, those that are entry related, and those that are transaction related. The `EntryException` class defines the method `getEntry` which returns the reference to the entry to which the exception pertains. More specific entry exception types may provide additional methods to obtain relevant information regarding why the exception was thrown.

The `TransactionException` hierarchy is similar. The `getTransaction` method returns the offending transaction reference.

Refer to tChapter 9, Class Files, for more information as well as the public API Javadoc documentation for details on what each exception type denotes and the conditions under which a particular method may throw a given exception.

Aliases

`SystemAliasEntry` can be used to reference other entries in the database (the *aliased entry*). Aliases may have children so that hierarchies can be created.

There are several rules regarding the creation of aliases:

- All aliases must be inserted into the *alias* namespace.
- Only aliases may be inserted into the *alias* namespace.
- The aliased entry must not be an alias.

At least for the first release of the JSD, these rules are enforced to simplify the transaction and locking issues associated with locking aliases and their aliased entries. They also help prevent cycles from being created using aliases. These rules may be relaxed in the future.

The aliased entry reference can be obtained using the `getAliasedEntry` method. This value is immutable and is set at alias construction time. It is possible to construct an alias that does not reference another entry, which only serves as part of the alias namespace hierarchy. For example, the root alias namespace entry (*/alias*) is an instance of `SystemAliasEntry`, but it does not alias another entry. It only serves as a vehicle for supporting children, which may or may not alias other entries.

An alias may end up referring to a deleted entry when the aliased entry is removed. In this case, the aliased entry is not garbage collected. Thus, aliases should not be used to reference dynamic entries unless the alias itself is deleted as well.

Multiple aliases may refer to a single entry. This does not mean, however, that each alias is a parent of the referenced entry. In fact, no reference is maintained by the aliased entry back to an alias. If the aliased entry is modified in any way, either directly or via a property related method call made on the alias, the associated event is generated with a filter string that names the entry using its natural parent hierarchy. No other events are produced on behalf of the alias itself.

Properties

The benefits of an alias are shown during property access. When invoking a property API method on an alias, one of three things can happen:

- If the alias does not refer to another entry, then all properties are local to the alias entry itself.
- Otherwise, property method calls are redirected to the aliased entry.
- If the aliased entry has been deleted, then an `EntryDeletedException` is thrown.

Method redirection is accomplished via the convenience methods of the aliased entry. In the first release of the JSD, this is a necessary trade-off in order to accommodate the simple transaction model.

Transactions

A transaction is the means by which one or more related entries in the JSD can be locked and accessed in an atomic manner. A transaction defines a set of entries. The `Transaction` class is implemented exclusively for write access to an entry. That is, a transaction object is needed for every write access to an entry; read access does not require a transaction object (although read access can impact transaction entry locking).

The JSD transaction model operates on a subtree of the database. A subtree is illustrated by the shaded entries in the figure below. The subtree is defined by a root entry and it extends throughout the hierarchy of its descendants to all leaf entries.

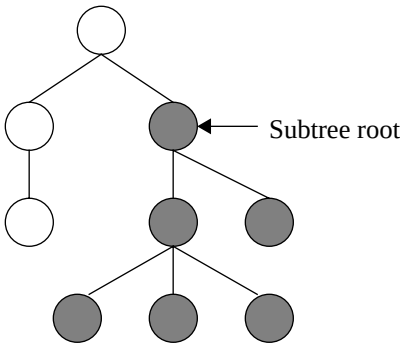


FIGURE 4-28 Subtree Definition

Transactions are defined to be either shared or exclusive. This maps to the mode in which the locks for the entries in the set are acquired. All public API methods that perform modifications (e.g. `insert`, `remove`, `addProperty`) require an exclusive transaction instance. Attempting to use a shared transaction will result in `TransactionTypeIncompatibleException` being thrown.

Methods that are only reading information (e.g. `getPropertyValue`, `getChildCount`) only require a shared transaction instance, though an exclusive transaction may be used as well for this level of access.

Transaction Definitions

The JSD provides a distributed, flat transaction model. A transaction must provide the four *ACID* properties:

1. *Atomicity.* Either all state changes associated with the transaction must occur or none occur.
2. *Consistency.* A committed transaction correctly transforms state.
3. *Isolation.* A transaction must ensure coherent state. Multiple transactions cannot modify state simultaneously, but rather serially.
4. *Durability.* After successful completion of a transaction, any state changes survive any subsequent failures (system crash, etc.).

On the client, local memory-based transactions are not durable (persistent). A distributed transaction model is required in order to support persistence. The `Transaction` class allows for defining a dependent distributed transaction identifier. The persistence implementation must make use of the dependent transaction to ensure the ACID properties are an end-to-end feature.

In this release of the network computer reference implementation for persistent entries, the D of ACID (Durability) will extend only so far as the successful completion of a write to the underlying Btree via the host operating system file system.

Transaction Factory

Transaction objects cannot be constructed directly. Rather, a factory is used to instantiate them based on criteria. The factory methods have a return type of the abstract class `Transaction`. Actual concrete transaction classes can change over time without affecting the use of the factory, or the public API.

All transaction based JSD public API methods refer to the type `Transaction` in their signatures. This is the same type returned by the factory. When necessary, a method may check at run time the actual type of the transaction. If it is unacceptable, then `TransactionTypeIncompatibleException` is thrown. Unfortunately, this limits the ability of the compiler to verify transaction types, but it provides the maximum flexibility to evolve the JSD transaction mechanism.

Figure 4-29 illustrates the transaction and public API interface and class hierarchy. Those in the cross-hatched area are public; those outside of it are private implementation classes.

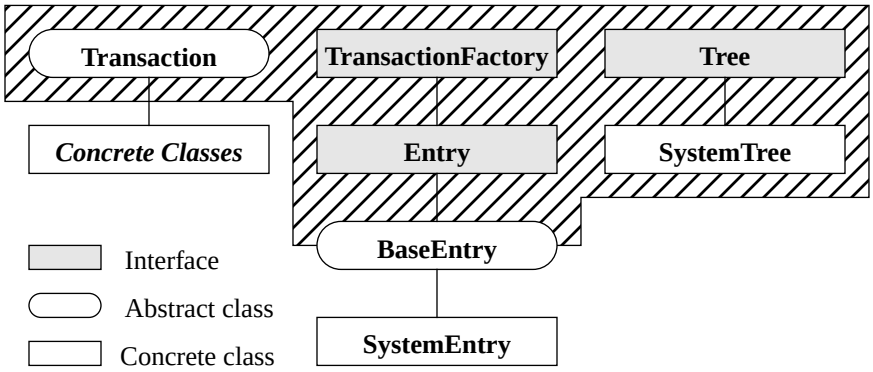


FIGURE 4-29 Transaction Related Classes

The `TransactionFactory` interface defines the following methods:

```
public Transaction createSharedTransaction() throws
SystemDatabaseException;

public Transaction createSharedTransaction(boolean noblock) throws
SystemDatabaseException;

public Transaction createExclusiveTransaction() throws
SystemDatabaseException;

public Transaction createExclusiveTransaction(boolean noblock) throws
SystemDatabaseException;
```

The `noblock` argument can be set to true, in which case the transaction is not constructed if it would have to wait for any lock within the specified set. Otherwise, the thread will block until all locks have been acquired, in either shared or exclusive mode, depending on the factory method invoked.

The `BaseEntry` class implements the `TransactionFactory` methods. The proper transaction concrete class is constructed using the entry upon which the method is invoked as the sub-tree root for the transaction.

While constructing the proper transaction, all of the entry locks within the specified set are acquired in the proper mode. As each is acquired, the transaction object notes the lock reference, and in the case of exclusive access, the lock itself maintains a reference to its owning transaction.

The current implementation of `Transaction` limits access to the entry lock to only the thread that constructed the owning transaction. This makes the entry lock serve with thread granularity. The owning thread is allowed to re-enter the lock, however. This policy could be modified in the future to allow simultaneous multi-threaded access (perhaps a thread group) to members of a transaction, but that is not a level of complexity needed for the first release of the JSD.

Transaction Usage Summary

Most methods in the JSD public API have both transaction based and non-transaction based versions. In the former case, a transaction is created behind the scenes when necessary. Exactly what is done depends on whether the method requires shared or exclusive access.

Convenience Method Usage

The non-transaction based methods in the public API are known as *convenience methods*. They simplify coding of applications because they create and use a transaction under the covers. However, while these methods may be convenient, they do present a couple of significant hazards:

- Atomicity across method calls is not possible. For example, invoking the convenience version of `getPropertyValue` on the same entry twice may not result in the same value being returned. Or, the property may have even been deleted by another thread between the two invocations.
- Invoking a convenience method from within transaction based code could result in a deadlock since the convenience method will attempt to acquire entry lock(s). This could cause the thread to block on itself.

If an application is not concerned with the above, then using the convenience method is appropriate.

Explicit Transaction Usage

At the time of transaction creation, via the factory, the locks of the defined entry set are acquired. If a remote persistence mechanism (i.e. JSD server) is involved, then end to end locking is achieved at this time. If no exception is thrown by the factory method, then the returned transaction is ready for use.

Using a JSD transaction involves three steps:

1. Create a `Transaction` object. The sub-tree root is the entry on which the factory method is invoked. During construction, all entries within the sub-tree are locked exclusively by the transaction. If a persistence mechanism is involved, either a local file, or a remote server, then all necessary locking is achieved end-to-end after the constructor returns without throwing an exception.

2. Perform operations on the locked entries. Exclusive transactions are required for operations such as inserting, disconnecting or removing, or modifying entry properties. Both exclusive and shared transactions may be used for reading information, e.g. `getChildCount, getPropertyValue, getChildEntries,` etc.

3. Commit the transaction by invoking the `commit` method or rollback the transaction via the `abort` method. These methods release all of the entry locks held by the transaction. The `abort` method rolls back all changes made using the transaction. This is an end-to-end procedure, so any dependent transaction required for persistence is also either committed or aborted appropriately.

A transaction is optimistic in that changes are actually made to the JSD during the course of the transaction, though these changes are not visible outside of the owning transaction. Only in the rare case where a transaction is aborted, does a potentially time consuming rollback occur.

Mechanisms

Entries, entry locks, and transactions are interrelated. The entry methods define critical sections and check to make sure the entry lock is used appropriately. This is enforced for all entries by the `BaseEntry` class. Subclasses of `BaseEntry`, implemented using the JSD entry subclass API, are unaware of transactions and locking.

Creating a Transaction

The transaction factory methods are used to create a transaction. The entry upon which the method is invoked is the sub-tree root of the set of entries to comprise the transaction. Figure 4-30 illustrates the creation of either a shared (see bold text where

differences matter) or exclusive transaction. The factory method constructs a transaction of the proper type. During construction, locks are acquired for the sub-tree and all of its descendents.

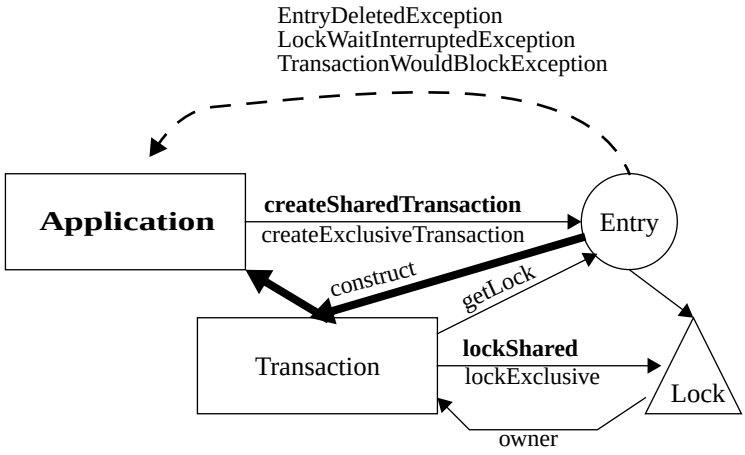


FIGURE 4-30 Transaction Creation

Each entry in the JSD maintains a reference to its lock object (an implementation of the `Lock` interface). The transaction obtains the lock reference for an entry by invoking its `getLock` method. Once the lock reference is obtained, the transaction will interact directly with the lock itself, and not the entry. The lock itself is acquired using the proper lock method, `lockShared` or `lockExclusive`, depending on the type of transaction being constructed. It is at this point where the thread constructing the transaction may block if the lock is not immediately available.

Once the transaction construction completes successfully, the reference to the transaction object is returned to the caller of the factory method. Note that any exceptions generated during transaction construction are also propagated to the caller of the factory method.

Using A Transaction

Once the transaction has been created, it is used by providing it to the transaction based public API methods. As shown below, the method implementation (in `BaseEntry`) obtains its own lock reference and verifies the entry is not deleted.

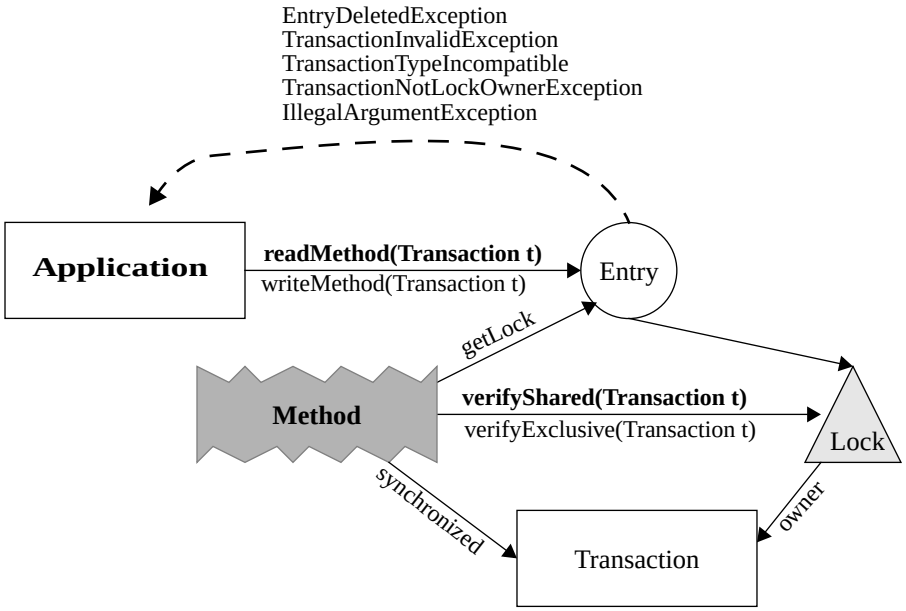


FIGURE 4-31 Using A Transaction

The method then synchronizes on the transaction so that the state of the transaction cannot change during the method's critical section. The entry lock is queried to verify that the specified transaction does indeed have access to the lock. Once verified, the API method proceeds to modify and / or obtain information on behalf of the transaction.

Terminating A Transaction

When a transaction is terminated, it releases all of the entry locks it holds and makes them available to other transactions. Successful termination is achieved by invoking the `Transaction commit` method. Any modifications made using an exclusive transaction are committed and events are produced to broadcast the updates. No such action is performed when a shared transaction is committed, as no updates are possible. However, it is still necessary to release the locks associated with both types of transaction.

An exclusive transaction may also be rolled back by invoking its `abort` method. The event queue is then used to undo all of the updates made using the transaction. The *abort* method of a shared transaction simply invokes its *commit* method so that the locks are released.

The `Transaction` object has a `finalize` method defined. If for some reason the application unreferences a transaction without first terminating it, the `finalize` method (invoked at garbage collection time) will abort the transaction in order to release locks held by the transaction. This also offers some protection against an application that terminates abnormally.

Exclusive Transaction Event Queue

When operating on published entries, the exclusive transaction class defines an event queue (JSE `OSEventQueue`) and a stack to record locks held by the transaction. While modifications are applied to locked entries, JSD events are maintained in the transaction key event queue. Key events are those that pertain to property changes, or the subtree root of a hierarchy change (point of insertion, deletion or removal).

When an exclusive transaction is committed, the key event queue is traversed. Any hierarchy change events are expanded by generating events for all descendents of the subtree root. Any state changes, manager inheritance etc. are applied at this time. The expanded event queue is then produced and made available to all consumers.

The event queue, and other ancillary `BaseEntry` fields, can be used to rollback the transaction. In this case, the event queue serves as a record of what operations were performed during the transaction and need to be reversed. Once rolled back, the events in the queue are discarded and are not produced.

Transaction Related Exceptions

The JSD exception hierarchy pertaining to transactions is illustrated in Figure 4-32. The `TransactionException` class provides the `getTransaction` method which returns the reference to the transaction object to which the exception pertains.

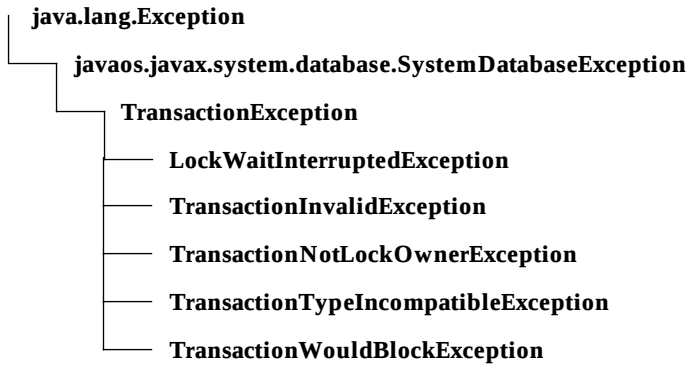


FIGURE 4-32 Transaction and Lock Exceptions

LockWaitInterruptedException

This exception indicates that while waiting to acquire an entry lock, the thread was interrupted (`java.lang.InterruptedException`). It is beyond the JSD to understand *why* the thread was interrupted, only to know that it *was*. Therefore, the transaction factory method simply throws this exception. All locks acquired by the factory method up to the time of the interruption are released.

TransactionException

This class denotes the base of the transaction specific exception class hierarchy. The `getTransaction` method can be invoked to return the reference of the transaction that inspired the throwing of the exception.

TransactionInvalidException

An invalid transaction was presented to a public API method. Once a transaction has been either committed or aborted it becomes invalid and can no longer be used.

TransactionNotLockOwnerException

The transaction presented to the public API method invoked on an entry does not own the lock for the entry. That is, when the transaction was created, it did not include the entry in its lock set.

TransactionTypeIncompatibleException

The type of the transaction is not suitable to the invoked public API method. All methods that may potentially modify an entry require and exclusive transaction. Methods that only read information accept either a shared or exclusive transaction.

TransactionWouldBlockException

A transaction factory method was invoked with the `noblock` argument set to true, and an entry was encountered within the subtree whose lock was not immediately available. All locks acquired up to that point by the transaction are freed prior to this exception being thrown.

Practical Transaction Usage

Simple one-shot entry access may be well served by simply invoking one of the public API convenience functions. Thus, the application program need not deal directly with transactions. Issues regarding the use of these methods have already been discussed earlier in this chapter.

Explicit transaction usage requires the application to create a transaction via the factory, make transaction based method calls, and then to finally commit or abort the transaction. Even though no modifications can occur, shared transactions must be committed at some point so that the associated locks are released.

When atomicity and isolation are important, it is necessary to use transactions explicitly in all but the most simple cases. For example, in order to enumerate all of the children of an entry, it is best to create a shared transaction by invoking the factory method on that entry. Then, the `Enumeration` returned by the `getChildEntries` method will return consistent results, as it is not possible for children to be unexpectedly added or deleted while the entry is locked. The same can be said for enumerating property names. No such guarantees are provided when the convenience methods are used.

A simple Java code usage template needs to encompass each step in the life cycle of a transaction. While most JSD related exceptions are a subclass of `SystemDatabaseException` it is possible to catch exceptions that are not, e.g. `IllegalArgumentException`, or some exception generated by the application itself.

In the simple case, if an exception is thrown, then the transaction is aborted. This may or may not be the case given a particular application program. But for the simple case, the following algorithm is recommended:

```
create transaction
try {
                    do something
                    commit transaction
} finally {
                    abort transaction
}
```

The exception must be created outside of the try/finally statement because it needs to be accessible in both the try code and the finally code. Also, any exceptions thrown during transaction creation can be handled by outer code, such as the calling method, or an outer try/catch/finally statement.

In the example above, if no exceptions are thrown by code executed in the try block then the transaction is committed and all is well. On the other hand, if any exception is thrown (from the `SystemDatabaseException` hierarchy or not), then the transaction is aborted.

The finally code is executed regardless of whether an exception is thrown. In the successful case, the transaction is aborted after it has already been committed. This has no effect: the commit is not reversed, and no additional exception is thrown.

Using Drafted Hierarchies

Entries in the drafted state are not subject to the same heavyweight locking constraints as are published entries. This assumes that only a single thread has access to the drafted entries it has constructed. This assumption doesn't work if a thread provides the drafted entry references to other threads, or if an entry (or hierarchy) had been disconnected by the published JSD hierarchy, because other threads may hold references to these entries.

Working with drafted hierarchies is particularly useful for network computer boot and user login processing using a JSD server, or local persistence mechanism. However, the advantage is just as valid for the application program that requires the creation of a significant hierarchy of entries (perhaps based on information read from a file).

Code Examples

Case 1

Access a single entry, performing multiple atomic updates:

```
entry1 = tree.findEntry(“/somepath”);
Transaction t = entry1.createExclusiveTransaction();
try {
                  entry1.addProperty(t, “name1”, value1);
                  entry1.addProperty(t, “name2”, value2);
                  entry1.removeProperty(t, “name3”);
                  t.commit();
} (finally) {
                  t.abort();
}
```

Case 2

Access three different entries and modify them in an atomic manner. In this example, *entry2* and *entry3* are descendents of *entry1*:

```
Transaction t = entry1.createExclusiveTransaction();
try {
                  entry1.addProperty(t, “name1”, value1);
                  entry2.addProperty(t, “name2”,
                  entry3.getPropertyValue(“name3”));
                  t.commit();
} (finally) {
                  t.abort();
}
```

Case 3

A more realistic example is an application that is updating its configuration information.

```
Tree t = SystemDatabase.getTree();
Entry me = null;
try {
                  me = t.findEntry(“/software/application/
     com.appsRus/killerapp1”);
} catch (EntryNameNotFoundException ex) {
                  // Oops, we have not been properly installed yet
                  me = install_self();
}
Transaction t = me.createExclusiveTransaction();
try {
                  me.addProperty(t, “background”, bgValue);
                  me.addProperty(t, “foreground, fgValue);
                  t.commit();

} finally) {
                  t.abort();
}
```

Navigation

Trees

A tree comprises the entire hierarchy from the root entry down through all of its descendants. Any arbitrary database entry may be a *designated tree root* and respond to a series of methods from the *Tree* interface. All descendants of the root are part of the tree. To provide some context, the *Tree* interface is used to defined methods common to all trees. These include methods for:

- Root Entry
- Current Entry
- Tree Populator
- Find and New (entries and aliases)

The tree's name is the same as the designated tree root, and belongs to a single namespace. Entries in the tree may be referenced by a pathname that is either relative to the root or a *designated current entry.*

Pathnames starting with a / name an entry relative to the tree root. All other pathname forms identify entries relative to the current entry. Methods to get and set the current entry are provided. A tree's current entry is analogous to the UNIX operating system's current working directory.

Tree Population

The *TreePopulator* interface provides a general mechanism for populating a given tree. In embedded versions of Java (in a cell-phone, for example), it may be desirable to populate sections of the database from a pre-defined *static table of strings*. A simple platform like a phone has a well-known number of fixed devices easily represented as an array of strings. More complex systems can implement a tree populator based on, for example, OpenBoot firmware, host operating system information.

Find and New

The *Tree* interface defines methods used to initiate pathname lookup within the tree (`findEntry`), as well as add new entries to the tree (`newEntry`,`newAlias`). These methods may do little more than invoke the `locate` method on the root entry.

Database PathNames

A *pathname* is a string that identifies an entry in the database. Pathnames are interpreted relative either to the superroot or a designated current entry within a namespace.

Navigating within the JSD

The JSD allows navigation via both entry references and pathname lookup. The former is done with the *Entry* interface `getParent` and `getChildEntries` methods. The latter is done by the `findEntry` method as defined by the *Tree* interface. The `Query` class is provided to perform a more complex search of the JSD based on entry names and property names. In the end, however, entry references are returned as the results of the query.

The `Entry getParent` method returns the reference to the entry's parent entry. An entry has only one parent. Entries in the drafted state may or may not have a parent. The `getChildEntries` method returns an `Enumeration`. The references to each of the children can be obtained this way. While these methods are commonly used inside of the database package itself, other more convenient mechanisms are available to find and create JSD entries.

The *Tree* interface defines the `findEntry`, `newAlias`, and `newEntry` methods. These methods accept a string representing the pathname of the entry of interest. A pathname is similar to a UNIX style filename, such as /usr/bin/rm. The exceptions `EntryNameNotFoundException`, `EntryNameExistsException`, and `EntryNameInvalidException` are thrown to indicate failure modes.

The *Entry* interface provides hooks for pathname lookup. These are the methods `locate` and `isBasename`. The `locate` method allows a database entry to indicate whether one of its children is part of the pathname, either as the targeted entry (the final component or base name), or as a component traversed to reach the targeted entry. A reference to an object of the class `LocateResult` is provided to `locate`. This object is modified during the course of pathname lookup.

Flexibility is provided within the JSD by *not* explicitly dictating what characters represent valid entry names and pathname component separators. The base `SystemEntry` class defines its pathname component separator as the forward slash (/). Thus, pathnames comprised entirely of `SystemEntry` objects are very UNIX-like. However, other subclasses of `BaseEntry` may choose a different component separator character, or restrict names in some fashion.

Figure 4-32 illustrates a subset of the JSD on a hypothetical x86 based client. In this example, all entries in this part of the database are `SystemEntry` objects. At boot time, after creating the superroot entry, a `SystemTree` object is instantiated to denote the top-level tree for the database. The root entry of this tree is the superroot entry.

A copy of this tree can be obtained using the `getSystemDatabase` method of the `SystemDatabase` class. In the example illustrated in Figure 4-33, the current entry of the tree is the entry indicated as /device/i86pc/isa, which represents the ISA bus in the machine.

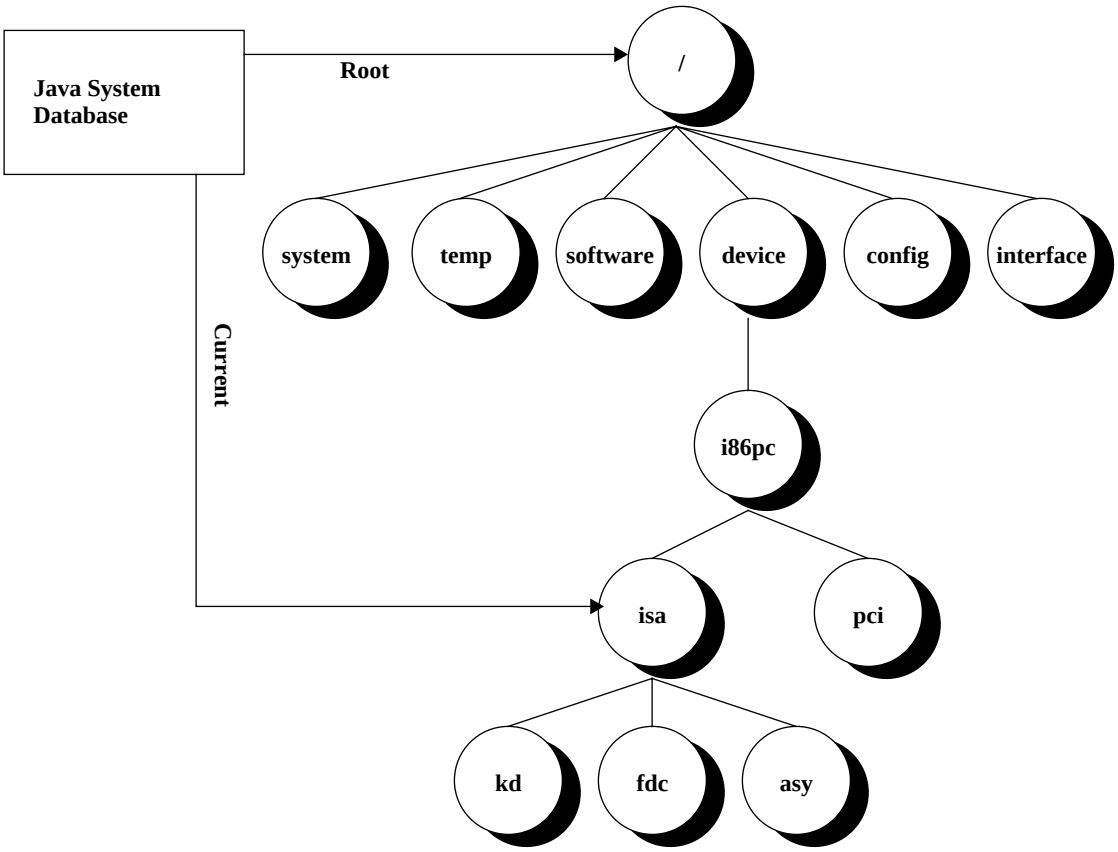


FIGURE 4-33 Navigation Example #1

In order to get the entry reference to the floppy disk controller on the ISA bus, the `findEntry` method of the system database tree object can be invoked with the pathname */device/i86pc/isa/fdc* or *fdc*, since the ISA bus entry is the current entry. pathnames not beginning with the separator character are with respect to the current entry defined in the `SystemTree` object.

The entry at which the pathname lookup begins is known as the *starting search point*. Pathname lookup proceeds as follows:

```
public Entry findEntry(Transaction t, String path) throws
                  SystemDatabaseException {
                      LocateResult lr = new LocateResult(path);
                      Entry e = getStartingSearchPoint(lr);
                      String nextComp;
                      while ((nextComp = e.locate(t, lr)) != null) {
                          if (nextComp.equals(“..”))
                              e = e.getParent(t);
                          else
                              e = e.getChildEntry(t, nextComp);
                      }
                  return e;
}
```

Using the example in Figure 4-33, given a pathname of */device/i86pc/isa/fdc*, the starting search point will be the superroot entry. On superroot, `locate` is invoked with the remainder of the path (*device/i86pc/isa/fdc*). Because there is a child of superroot with the name of device, `locate` succeeds and the entry reference to the matching child is returned by invoking the `getChildEntry` method. The pathname then becomes i86pc/isa/fdc. The `locate` method of the device entry is then invoked and so on. The `locate` method of the entry named isa is invoked to locate its child named fdc. Finally, `locate` is invoked on the entry name fdc and null is returned. The path is now exhausted, indicating that we succeeded in matching the final pathname component.

If at any point `getChildEntry` fails (throws an exception), then the pathname component does not name an existing child of the entry. The lookup then fails by the exception being propagated up to the caller of `findEntry`.

The `locate` method, in addition to indicating whether a pathname component (in the `locateResults` object) remains, also updates the pathname indexes by advancing past the last component that matched. Therefore, the above while loop will end eventually. The benefit of this mechanism is that the implementation of the `BaseEntry` subclass defines what comprises a path component. This policy is enforced by the parent on behalf of its children.

For example, given the hypothetical class `NFSEntry`, which represents a mounted NFS file system, how would navigation behave differently for this entry (gray) in Figure 4-34. In this example, the current entry of the system database tree is the volume entry, so the NFS mount point entry can be reached as either */interface/volume/mntpt* or just *mntpt*.

To find the entry /interface/volume/mntpt/dist/bin/prog, the procedure begins as in the first example. However, when the `locate` method is invoked on the mntpt entry, it is provided with the path dist/bin/prog. It can use the remainder of the path to obtain information from the NFS server. The `NFSEntry` object could then create a database entry that refers to the specified file. Or, the `NFSEntry` itself could maintain some state, and represent the file itself (perhaps via a Hash table entry based on the remaining path).

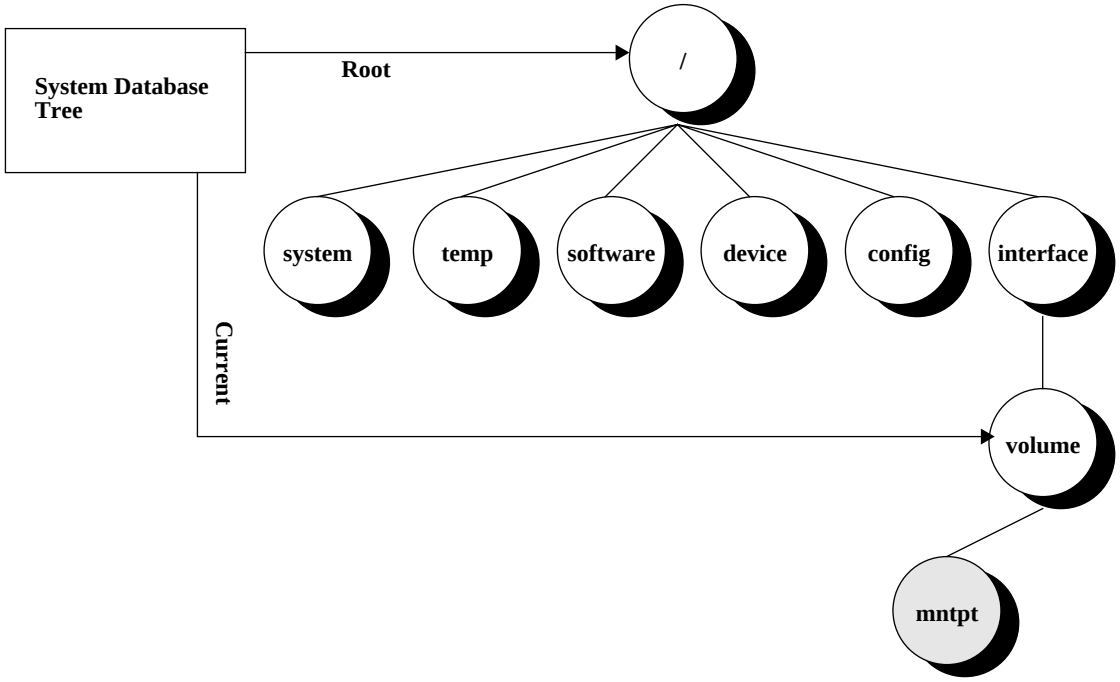


FIGURE 4-34 Navigation example #2

It is easy to see that a given entry implementation can be highly customized. The `locate` method of a specialized entry could be given a path of *machine1.com|machine2.com:STATUS*, and it will know how to break down the string and, for example, query the two machines for status and then instantiate a database entry to represent the status.

When creating a new entry based on a pathname, the base name (last component) of the string is the name to be used for the name of the new entry. This is true for `SystemEntry` at least. Since during the lookup process, the pathname syntax of individual entries is unknown, the `SystemTree newEntry` method can perform the lookup to the point of the parent entry. If no child entry matches the remaining path, then a new entry object is instantiated (via the parent's manager) and its name is that of the remaining path.

However, a remaining path of, say, xxx/yyy is invalid for `SystemEntry` because it contains the component separator. Since `SystemTree` does not know what the separator is, or if there are other limitations on entry names, it invokes the parent entry's `subVerifyName` method. This method is part of the `BaseEntry` subclass API and indicates whether or not the pathname component is appropriate for a child entry. If it is, then the parent's manager can be used to create a new instance to be inserted as a child.

The pathname lookup approach described is somewhat more complicated and costly to implement then simply enforcing seemingly arbitrary JSD rules for all entries, but it provides a great deal of power and flexibility.

Searching the Database

The JSD defines the classes called `Query` and `PropertyQuery` which search the database for entries that match designated criteria. The matching criteria can include the entry name or any property name

Query objects are constructed by providing a starting search point (entry) and search criteria. A query supports Java's enumeration interface to walk entries matching the criteria. Additional methods to reset and backup the search are also provided. A JSD query includes the following:

- Name (String)
- Scope of search
- State of search
- Entry of last match
- Entry of current match

Search Criteria

`Query` class constructors allow for the specification of name strings to be matched against all entries defined within the search scope. The `PropertyQuery` constructors provide the same functionality for searching for property names.

Search Scope

The scope of the search can be self, parent, siblings, children, or descendants.

self

Only the starting search point entry is searched for the entry or property names.

parent

Only the parent of the starting search point entry is searched for the entry and property names.

siblings

All of the siblings or the starting search point entry, including the starting search point entry itself, are searched for the entry and property names. The search descends no further.

children

Only the immediate children of the starting search point are searched for the entry and property names.

descendants

All descendants of the starting search point entry (entire sub-tree hierarchy) are searched for the entry and property names.

Search Results

Methods are provided by the `Query` and `PropertyQuery` classes to obtain the entry reference of each match. A standard Java `Enumeration` interface is also implemented.

JavaOS Configuration Tool (JCT)

The JCT is a system application used to administer and configure software for JavaOS for Business network computers. It consists of an HTML/Applet based user interface, providing commonly used administrator functions for configuring machines, groups, users, and software. It consists of several components: Navigation, Configuration Environment [JCE], Bean UI, JAR Services, and JSD System which communicate with each other as shown in Figure 4-35.

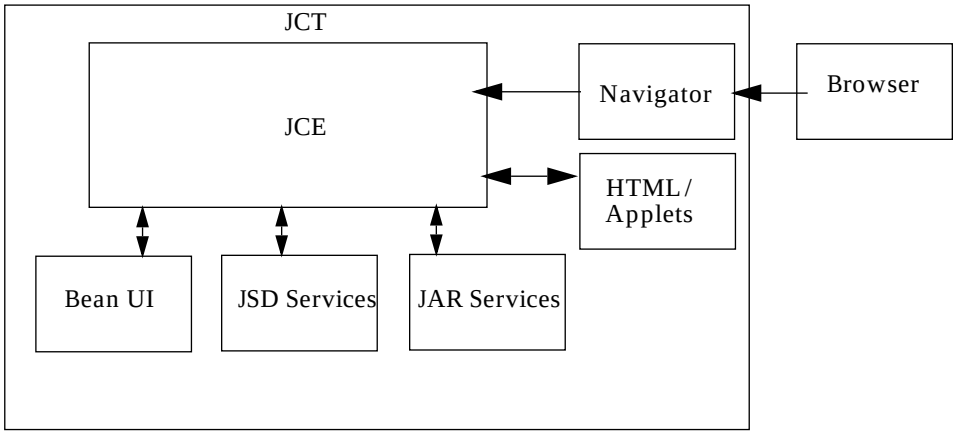


FIGURE 4-35 The JCT's Relationship with the JCE

The JCT Navigator

The Navigator is an administrator interface that starts when the Java browser is pointed at `STARTJCL.HTML`. When the administrator narrows the configuration selection down to a terminal selection, the Navigator is invoked. The Navigator parses parameters and initializes the JCE, which in turn launches the appropriate JAR resources.

JCE

The JCE is not a visible part of the user interface. It provides services to the JCT Navigator, to the JSD, and the JAR files, and provides a user interface for displaying Bean properties.

Bean User Interface

The Bean user interface displays Bean properties. It is invoked by the JCE for any configuration requests whose target resource is a Bean. It receives as input a JAR File loaded by the JAR services.

JAR Services

This component of the JCT abstracts JAR files and their contents to the JCE. It opens, closes, enumerates, and determines JAR resource contents. It also contains a class loader to load the archived classes contained in the designated JAR file.

JSD System

This component of the JCT provides interfaces between the JCE and the JSD Server, by which the JCE can load and store pointers to JAR resources.

5

JavaOS for Business System Events (JSE)

Introduction

This chapter describes the JavaOS for Business Event System.

- Introduction
- Role of the Event System
- Summary of the Event System
- OSEvents
- Event System Broker
- OSEvent Producers
- OSEvent Consumers
- Sample Device Driver Event

All operating systems require a robust and flexible communication mechanism from which to build cooperating layered services. The role of the JavaOS for Business Event System is to manage, monitor, and update the exchange of information between services, such as file systems and device drivers. The OSEvent classes provide flexible, layered synchronous and asynchronous communication between the OS and its services.

The Event System is *not* intended as an application-level event system. The Event System operates at the lowest levels of the OS to transfer information to higher levels in the system. At higher levels, events may be translated into Java Developer Kit (JDK) events.

Figure 5-1 illustrates the core components and interfaces. The Event System resides within JDI.

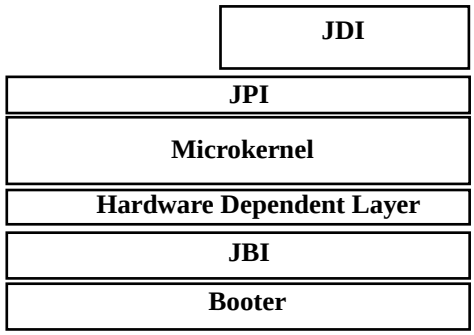


FIGURE 5-1 JavaOS for Business Layers

The JDI can be further separated into a number of components, as illustrated in Figure 5-2. The Event System is one of these components.

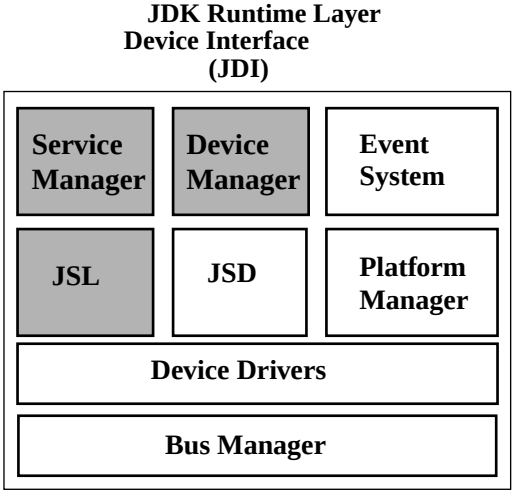


FIGURE 5-2 JDK Runtime Layer JDI Components

NOTE: Shaded areas are not documented in this manual.

Events result from applications, services, and OS components that generate information needing to be passed on to other similar programs in the operating system. A program that generates an event is called a *producer*, and a program that receives event information is called a *consumer.* Programs can perform dual role, acting as both a producer and a consumer. To pass an event from a producer to a consumer, the event *handler* method is invoked by the Event System.

The coordination and management of all information passing between event producers and event consumers is performed by the *Event System broker*. The broker maintains a separate *master producer list* and a *master consumer list* on the system. In addition, the broker is responsible for linking the two lists. This is done by means of queues.

Each producer on the master producer list has *consumer queue*s. One consumer queue exists for each type of event class. The priority given to each consumer on the queue is determined from a variety of consumer *rule sets*. For example, a rule set exists to establish ordering between consumers of the same class of event. Another rule set determines matching between a consumer and an event using a *filter*, such as a JSD pathname string. Another rule set determines how an event consumer will *consume* the event, that is, will the consumer share the event, be competitive with other consumers, or demand exclusive rights to the event.

Rule set specifications, as well as other information derived from the OSEvent base classes, enable the Event System broker to effectively and efficiently manage the exchange of event data between producers and consumers.

The Event System serves as a general purpose event system. The Event System's primary interaction is between the JSD, the Service Loader (JSL), and the system console.

The following sections provide more detailed information of the workings of the Event System.

Figure 5-3 illustrates the direction Event System information passes between layers within JavaOS for Business.

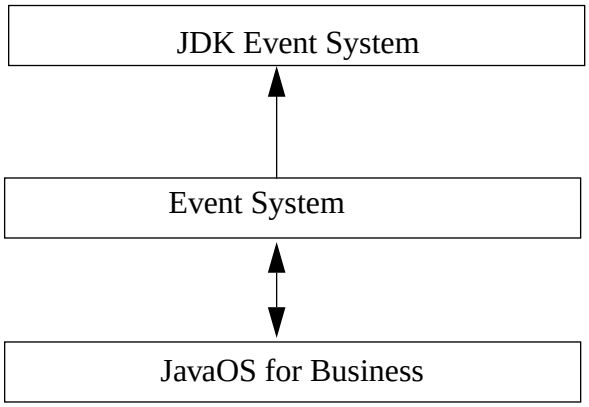


FIGURE 5-3 Event System Information Flow

The Event System design is based on four key elements:

- OSEvents - typed information passed between operating system layers
- OSEvent Producers - the source of events
- OSEvent Consumers - the recipients of events
- Event System Broker - the manager of information between all producers and consumers

OSEvents

An event is a typed (primitive or reference) piece of information, as defined by a class, that is exchanged between software objects. The simplest event contains a single piece of information that references the source of the event.

The source of the event is called the event *producer*. The recipient of the event is called the event *consumer*.

All system events derive from the `javaos.javax.system.events.OSEvent` base class and are constructed with a reference to their producer. The base class contains methods for:

- returning a reference to the event producer
- copying the private data of one event to another event
- returning an indication as to the kind of consumption

Event System Broker

The role of the Event System is to serve as a broker between event producers and event consumers. The term *`event system`* refers to the class (or static) methods and data defined in the `OSEventProducer` abstract base class. These class methods and data provide the standard broker behavior for all concrete subclasses of `OSEventProducer`.

By acting as a broker, the Event System separates producers from consumers and removes the burden from event producers of tracking and managing the objects that consume their events. In addition, the broker updates production and consumption matching as new producers and consumers are added or removed.

For example, a device driver that produces an event containing newly available data simply builds the event and passes it along to the Event System to distribute to consumer objects subscribed to that type of event. The device driver does not need to keep a list of event consumers, nor does it need or interact directly with consumers. The Event System broker manages the exchange of all information between producers and consumers.

Figure 5-4 illustrates the relationship between producers, consumers, and the Event System broker.

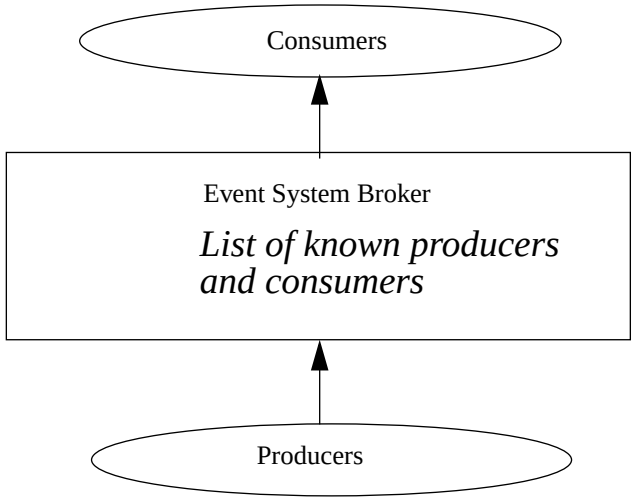


FIGURE 5-4 The Event System Broker

Subscription and Registration

Producers *register* once with the Event System as a source of events of a certain type. Event producers are created with a list of associated event kinds. In order for a producer to make itself known, its registration information is placed on the *master producer list*.

Consumers also *subscribe* once with the Event System to events of a certain type. In order for a consumer to make itself known, its subscription is placed on the *master consumer list*. Event consumers are also created with a list of associated event kinds. For producer and consumer event matching, a consumer is added to one or more producer consumption queues.

The Event System supports two kinds of consumer subscription. The first kind subscribes a consumer with all producers of a specified kind of event. Multiple producers may produce the same type of event. The second kind of subscription subscribes a consumer with a single producer.

Once registration and subscription take place, all timing issues are handled by the Event System broker.

Figure 5-5 illustrates various event subscription relationships.

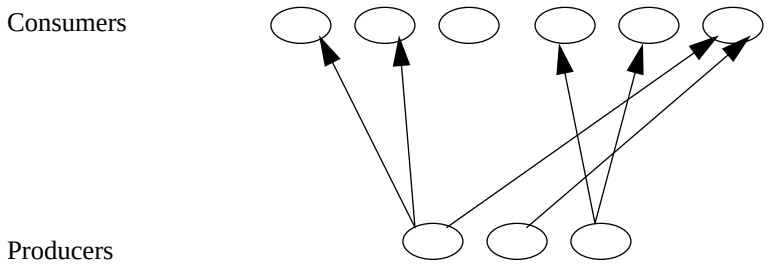


FIGURE 5-5 Event Subscriptions

Producers and consumers can undo registration or subscriptions if they no longer want to participate in the Event System. The Event System constantly updates its registration and subscription lists as the total set of producers and consumers changes. The lists of active producers and consumers may be enumerated by taking a snapshot of the currently active lists.

Registering a Producer with the Event System

Registering a producer is accomplished by invoking a producer's `register` method with a boolean value to indicate whether this producer is the sole producer of all its associated events.

Finding Registered Producers

Finding a registered producer is a two-step process. First, a consumer asks the Event System for an `OSEventProducerEnumerator` object by invoking the static `producers` method. Second, the consumer enumerates the master producer list by invoking the `OSEventConsumerEnumerator` object's enumeration method, `nextElement`.

Unregistering a Producer

Producers can be unregistered using a static method in the `OSEventProducer` class called `unRegister`. Once a producer unregisters itself from the Event System, it is removed from the master producer list and its consumption queues are emptied.

Subscribing a Consumer with the Event System

Subscribing a consumer is accomplished by invoking a static method in the `OSEventProducer` class called `subscribeConsumer`. This method adds a consumer to the Event Systems master consumer list, and then connects the consumer to the appropriate producer consumption queues. The consumer consumption rules (shared, competitive, or exclusive) and ordering rules (ordered and unordered) determine consumer positioning on the producer consumption queue.

Subscribing a Consumer with a Single Producer

Event consumers can subscribe to individual producers. In this model of subscription, the consumer finds the registered producer and then invokes that producer's `connectConsumer` method.

Finding Subscribed Consumers

Finding a subscribed consumer is a two-step process. First, an object asks the Event System for an `OSEventConsumerEnumerator` object by invoking the static `consumers` method. Second, the consumer enumerates the master producer list by invoking the `OSEventProducerEnumerator` object's enumeration method, `nextElement`.

To determine the quantity of consumers subscribing to any one producer, each producer object will return an `OSEventConsumerEnumerator` for each class of existing events.

Unsubscribing a Consumer

Consumers can be unsubscribed using the `unSubscribeConsumer` method. Once a consumer unsubscribes itself from the Event System, it is removed from the master consumer list and from any producer consumption queues.

Master Lists

The Event System keeps two master lists to facilitate event monitoring and matching:

- Master Producer List
- Master Consumer List

Updates to the master lists are performed continually as new producers and consumers are added and removed.

Master Producer List

Each producer registered with the event broker is on the master producer list. Each producer has one or more consumption queues that contain a list of consumers wanting to receive events from that specific producer. The producer has one consumption queue for each type of event.

Master Consumer List

Each consumer subscribing to the event broker is on the master consumer list. In addition, consumers reside on one or more producer consumption queues. The Event System provides a base `OSEventProducer` class that manages the queue of consumers waiting for event delivery from a specific producer.

OSEvent Producers

Event producer is the name given to objects that are the source of events. Event producers derive from the `OSEventProducer` abstract base class. A single producer is capable of producing more than one class of event. To actually produce an event of a particular class, a production method is invoked that specifies either 1) the class of event to produce, or 2) an event object to produce.

Exclusive event producers

The Event System lets a producer claim itself as the sole producer of an event. This feature ensures the secure production of events from components such as the JSD.

OSEvent Consumers

Event consumer is the name given to objects that receive events. Consumers are created by implementing and extending the `OSEventConsumer` interface. This interface contains five standard methods for receiving events, including:

- returning a consumer name
- returning the class of a desired event or events
- returning a boolean value to identify matching instructions
- recognizing a string for filtered events
- specifying if a handler disallows subsequent sharing of events

Each event class specified by a consumer can optionally match any sub-class of that event as well.

Consumer Rule Sets

Three sets of rules exist for consumer events:

- Consumption rules
- Ordering rules
- Matching rules

Consumption Rules

The Event System recognizes three different kinds of event consumption:

- Shared Event Consumers
- Competitive Event Consumers
- Exclusive Event Consumers

Each sub-class of `OSEvent` can define whether events may be shared among consumers, and whether consumers can compete for events. The default methods in the `OSEvent` class allow for shared consumption of events and disallow exclusive or competitive consumption.

Shared Event Consumers

Most event consumers are willing to share events with other consumers. There is no limit to the number of consumers that may share an event. The same event is given to each consumer in the consumption queue.

Competitive Event Consumers

The Event System allows consumers to compete for the right to completely consume the event. That is, prevent others further down the consumption queue from receiving the event, effectively *eating* the event.

To cause the broker to take an event, the handler method returns a special value to the event system that short-circuits the event system's delivery process, thus preventing downstream consumers from receiving that same event.

Exclusive Event Consumers

Some consumers want to be the sole recipient of an event. With exclusive consumption, consumers are rejected by the event system if an existing consumer already subscribes to an event desired by the exclusive consumer.

Ordering Rules

The Event System recognizes two classes of consumer event ordering:

- Ordered Consumers
- Unordered Consumers

Each sub-class of `OSEvent` can define whether consumers may prioritize their place in the consumption queue. The default methods in the `OSEvent` class allow for unordered consumers and disallow ordered consumers.

Ordered Consumers

The Event System lets consumers order themselves with respect to other consumers of a similar class of event. This feature allows for chains of ordered event consumers, giving each consumer the ability to alter the event data before it arrives at the next consumer. For example, a compression package could install itself at the beginning of the event consumption queue and thus compress the event data for all subsequent consumers.

When attempting to insert an ordered consumer in the queue of existing consumers, the event system uses the `OSEventOrderedConsumer` interface to check for conflicts. This verification is a two-step process.

During the first step, the interface checks the ordering rules of the existing consumers against the candidate consumer to determine if adding the candidate before or after each consumer is permitted. If any existing ordered consumer returns a *`false`*, indicating a conflict with positioning, the candidate consumer is rejected. If all existing ordered consumers return a *`true`*, then the verification proceeds to the second step.

During the second step, the interface checks the rules of the candidate consumer against the existing ordered consumers to determine if any positioning conflicts exist. If any existing ordered consumer returns a *`false`*, indicating a conflict with positioning, the candidate consumer is rejected. If all existing ordered consumers return a *`true`* then the candidate consumer is added to the consumption queue in the order originally requested.

Figure 5-6 illustrates the ordered event verification steps.

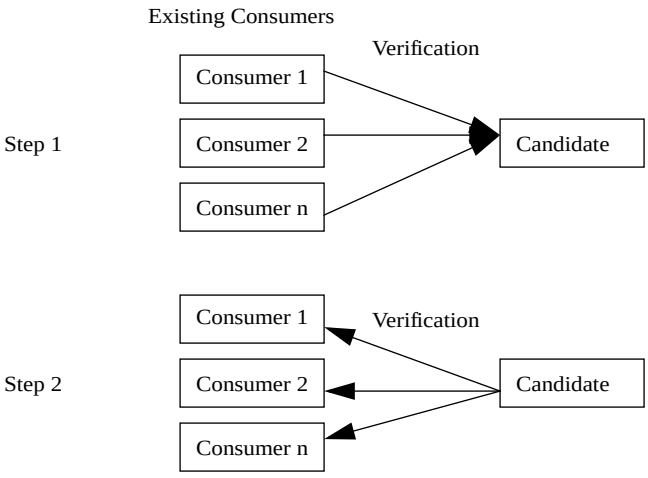


FIGURE 5-6 Ordered Event Verification Steps

Once an ordered consumer is placed in the consumption queue, its ordering rules are given priority over future consumers. That is, if the ordering rules of an existing consumer conflict with a candidate consumer, the existing consumer wins. Candidate consumers that cannot have their rules met are rejected during the event system verification steps.

Unordered Consumers

Unordered event consumers have no ordering preference when receiving event information. However, a verification is required to determine if the unordered consumer conflicts with any existing ordered event consumers.

When attempting to insert an unordered consumer in the queue of existing consumers, the event system uses the `OSEventOrderedConsumer` interface to check for conflicts. This verification is a one-step process.

During the verification step, the interface checks the ordering rules of the existing consumers against the unordered candidate consumer to determine if additions are permitted before or after each ordered consumer already in the consumption queue. If any existing ordered consumer returns a *`false`*, indicating a conflict with positioning, the unordered candidate consumer is rejected. If all existing ordered consumers return a *`true`*, then the unordered candidate event consumer is placed in the consumption queue in the order in which it was received.

Matching Rules

The Event System recognizes three kinds of consumer event matching:

- Exact Matching
- Sub-class Matching
- Filtered Matching

The Event System lets consumers specify matching requirements. Each sub-class of `OSEvent` can define event matching as either an *exact match* or a *sub-class match*, and as a *filtered match*. The default methods in the `OSEventConsumer` class allow for exact matching between a consumer and an event, and disallow sub-class matching between a consumer and an event. The consumer determines whether exact matching or sub-class matching is invoked.

Filtered matching lets a consumer use an event pathname string to target events in specific locations in the JSD, for example. The default methods in the `OSEventConsumer` class allow for matching between a consumer pathname string and an event. The event class definition determines whether filtered matching is invoked.

Exact Matching

This method gives each consumer the ability to indicate that event matching can take place only at the class level.

When attempting to determine matching, the event system uses the `OSEventConsumer` interface to determine whether the consumer wants an exact instance of an event class. The consumer returns *`true`* to indicate that the matching is exact.

Sub-Class Matching

This method gives each consumer the ability to indicate that event matching can take place at any level, from the class level down through the sub-class levels.

When attempting to determine matching, the event system uses the `OSEventConsumer` interface to determine whether the consumer wants a sub-class instance of an event class. The consumer returns *`false`* to indicate that the matching is sub-class.

Figure 5-7 illustrates the difference between exact matching and sub-class matching.

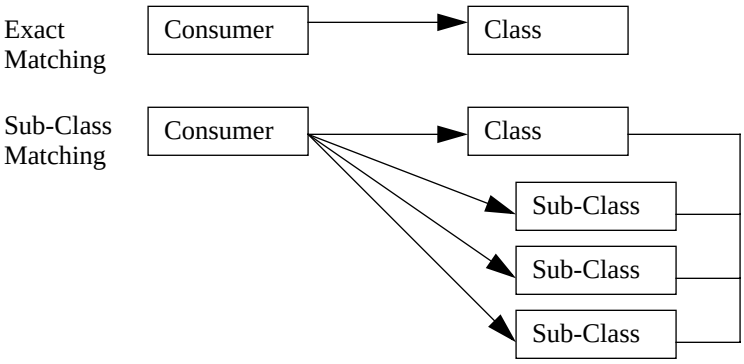


FIGURE 5-7 Exact Matching and Sub-Class Matching

Filtered Matching

Filtered matching can take place between a consumer and an event by use of a filter. While numerous filters exist, the most common filter is a JSD pathname string.

Filtered matching is determined through the `getFilter` method in both the event and the consumer. To compare filter strings, the event system broker first invokes `getFilter` to read the string from the event. Next, the broker invokes `getFilter` from each consumer in the producer's consumption queue. If the consumer string is present and *starts with* the event string, the event is given to the consumer. If the consumer string is null, the event is always given to the consumer.

The matrix in Figure 5-8 depicts the various consumer and event filtered matching scenarios.

	Event contains a string	**Event contains a null**
Consumer contains a string	Conditional. If True (match): give to the consumer. If False (no match): no error generated	False (no match): error generated to the producer
Consumer contains a null	True (match): give to the consumer	True (match): give to the consumer

FIGURE 5-8 Consumer and Event Filtered Matrix

In particular, JSD events are filtered to let consumers specify pathname strings for event consumption from specific areas in the database. If a consumer uses a filter string containing only a directory pathname, then all subdirectories within the directory are matched. For example, if an event producer is located in the */X* directory, and a consumer specifies a filtered string of */X*, then the consumer receives all events contained in the root directory, as well as any subdirectories in the root directory. Conversely, if the producer's location is */X/Y/Z/1/2/3* and a consumer specifies a filtered string of */X/Y/Z/1/2/3*, then the consumer does not receive any events appearing above this subdirectory level.

Presently, most events are filtered. An example of an event that is not filtered is the keyboard input activity event, `OSInputActivityEvent`.

Threading

The Event System supports a variety of threading scenarios. The event may be processed by a consumer using the thread of the producer, a *synchronous* scenario. Or, producers and consumers may use one or more threads to garner a desired level of *asynchronous* behavior.

With synchronous threading, no thread switching takes place. The consumer uses the thread of the producer. With asynchronous threading, the original event thread is replaced by the thread in the consumer queue.

Figure 5-9 illustrates the possible combinations of consumer and producer threads.

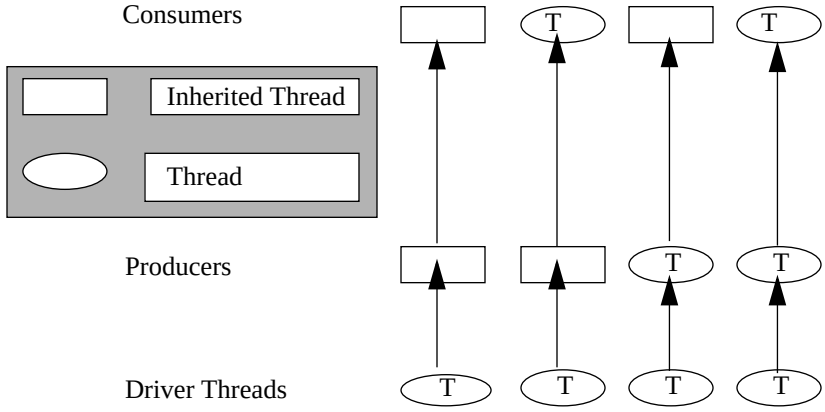


FIGURE 5-9 Producer and Consumer Threading Scenarios

The following threading scenarios are possible between event consumers and producers:

- The event is passed to the producer and on to the consumer for processing in a single thread.
- The event is passed to the producer and on to the consumer, at which point the consumer switches threads to process the event.
- The event is passed to the producer, at which point the producer switches threads before passing the event to the consumer.
- The event is passed to the producer, at which point the producer switches threads before passing the event to the consumer, at which point a second thread switch is done to process the event.

Sample Device Driver Event

The following example describes, in general, the interrelationship between an event, the event system, and the various operating systems components involved in the process of locating and acknowledging a new device.

In this scenario, a PCI expansion board has been added to a network computer. In order for the board to function, a logical connection needs to be made between the board and the appropriate device driver. As a result, the operating system must first recognize the new board, then pass the pertinent information to various system services. Once this process is complete, the new device can be matched with the appropriate device driver and function properly on the network.

The procedure to match a new device with its driver impacts the JSD, the JSL, and the Event System.

The scenario described above and the subsequent steps detailed below, are predicated upon the assumption that system has booted, and the JSD, JSL, including the device manager, among other services, are loaded. Furthermore, once the system activates, the JSD, during its initialization, registers with the Event System as an exclusive producer of `EntryInsert` events. In addition, the JSL, during its initialization, subscribes to the Event System as a consumer of `EntryInsert` events.

The step numbers listed below correspond to the steps found in Figure 5-9.

Step 1: The device manager scans the JSD device tree looking for busses. For each bus discovered the device manager calls the associated bus manager `probe` method. In this example, the `probe` method activates the PCI expansion bus manager.

Step 2: Once activated, the PCI expansion bus manager searches the physical bus to determine if all connected devices are registered in the JSD device tree. At this point, this specific PCI expansion board is not yet registered in the tree.

Step 3: The PCI expansion bus manager recognizes the board is new, and that it is not registered in the JSD device tree. As a result, the bus manager inserts a JSD device entry (child) in the tree, under the PCI entry (parent).

Step 4: In acknowledgment of the new device entry insertion for the PCI board, the JSD alerts the system by generating an `EntryInsert` event. Once the event is produced, it is passed to the Event System.

Step 5: The Event System receives the JSD `EntryInsert` event and, through the Event System broker, identifies the JSL as a subscriber of `EntryInsert` events. Then, the Event System calls the JSL's `OSEventConsumer` handler to pass the event to the JSL.

Step 6: The JSL receives the event and (using the device name in the event) searches for a match between this event and all of the driver business cards. Once the JSL matches the device name with the correct driver business card, the PCI driver is assigned to the PCI device. At this point, the PCI expansion board is now recognized as an active device on the network.

Figure 5-10 graphically depicts the process flow of the sample device driver event.

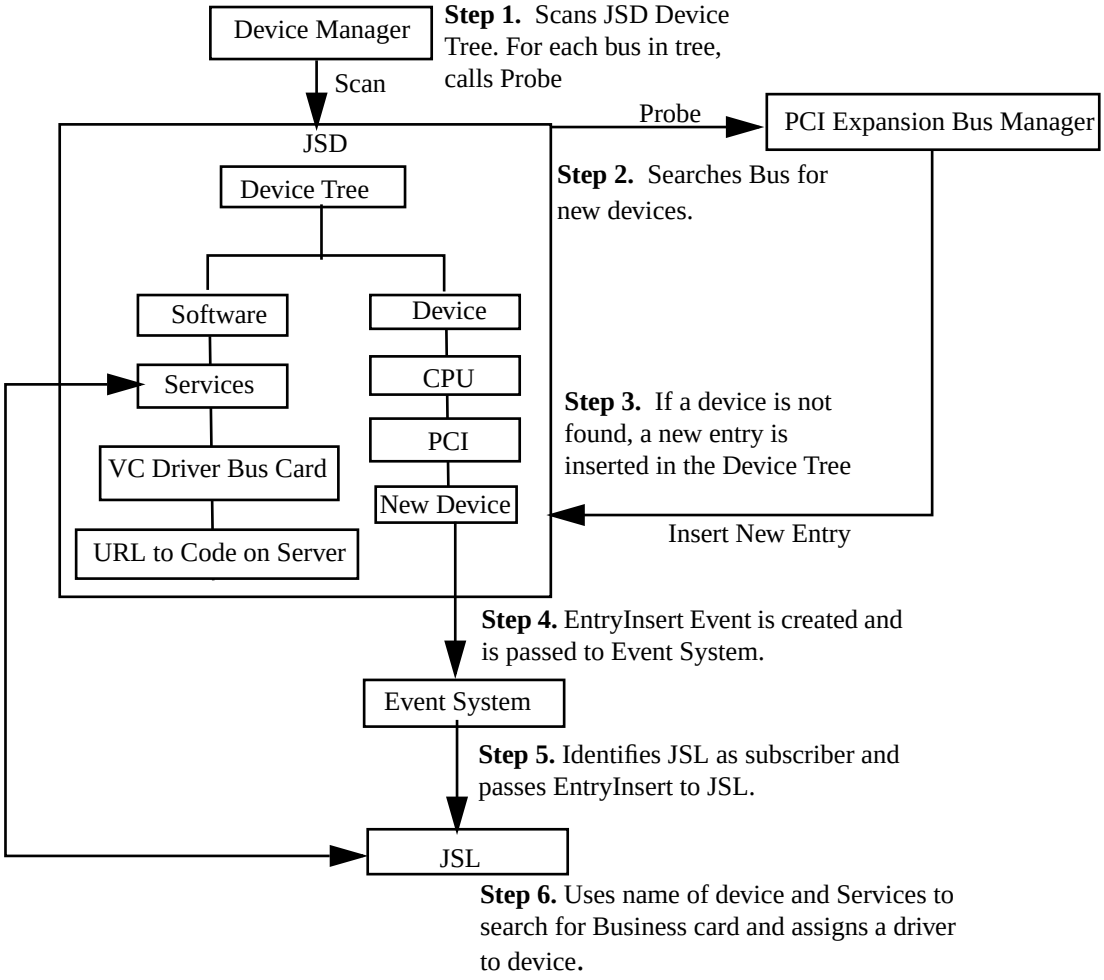


FIGURE 5-10 Sample Device Driver Event Flow Chart

6

JavaOS System Loader (JSL)

The JavaOS System Loader (JSL) manages the loading and unloading of system software components known as *services*. A service can encapsulate various operating system functionality such as device drivers, network protocols, and file systems. Services are loaded into the client platform on demand by the JSL from various containers such as JAR, ZIP, and class files.

The mission of the JSL is to minimize platform memory footprint by only loading and activating system services when needed. As soon as the service is no longer required, the JSL unloads it and allows the JVM garbage collector to reclaim memory occupied by the service.

Overview

The majority of system services are stored and loaded from a network computer associated server. This allows a central administrator to easily update and track services, using a product such as JavaSoft Java Server to drag and drop services in and out of the Java System Database (JSD).

JavaOS for Business is composed of many services loaded from the server. JavaOS for Business for a network computer is comprised of drivers for ethernet, mouse, display, and serial at a minimum. The network protocol stack in the JavaOS for Business software is also packaged as a service. The only portions of the software that are not packaged as services are those components that make no sense to load or replace, like the JSD or JSL itself.

Services can be used directly by an application or by other services in a layered fashion. The example in the following figure illustrates a sample I/O path from application to file system through layers of disk and SCSI services:

.

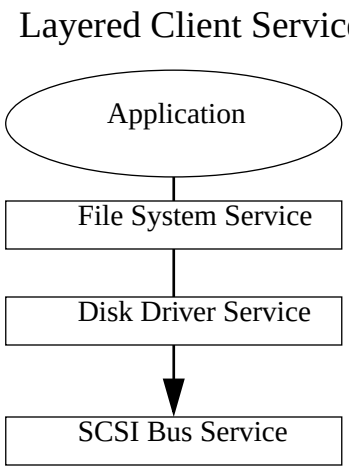


FIGURE 6-1 Sample I/O Path

This chapter discusses the JSL, services, and how services are used to implement traditional system software components such as device drivers. The reader may also want to refer to Chapter 2, JavaOS Device Interface, for an explanation of how the JDI uses the JSL to load and unload drivers.

The focus of the JSL is the activation of services (Java packages) that run on the client requesting the service. The JSL is not intended to access remote services. Network service location protocols such as SLP (Service Location Protocol) and JAW (Java Agent-Ware) are better suited to remote services.

The JSL is platform-independent. It requires access to file system services, the JSD, and the `java.system.events` package.

The Java service classes and loader work closely with the JSD to load, start, stop, and unload services. The following figure illustrates the various components that constitute the Java services architecture.

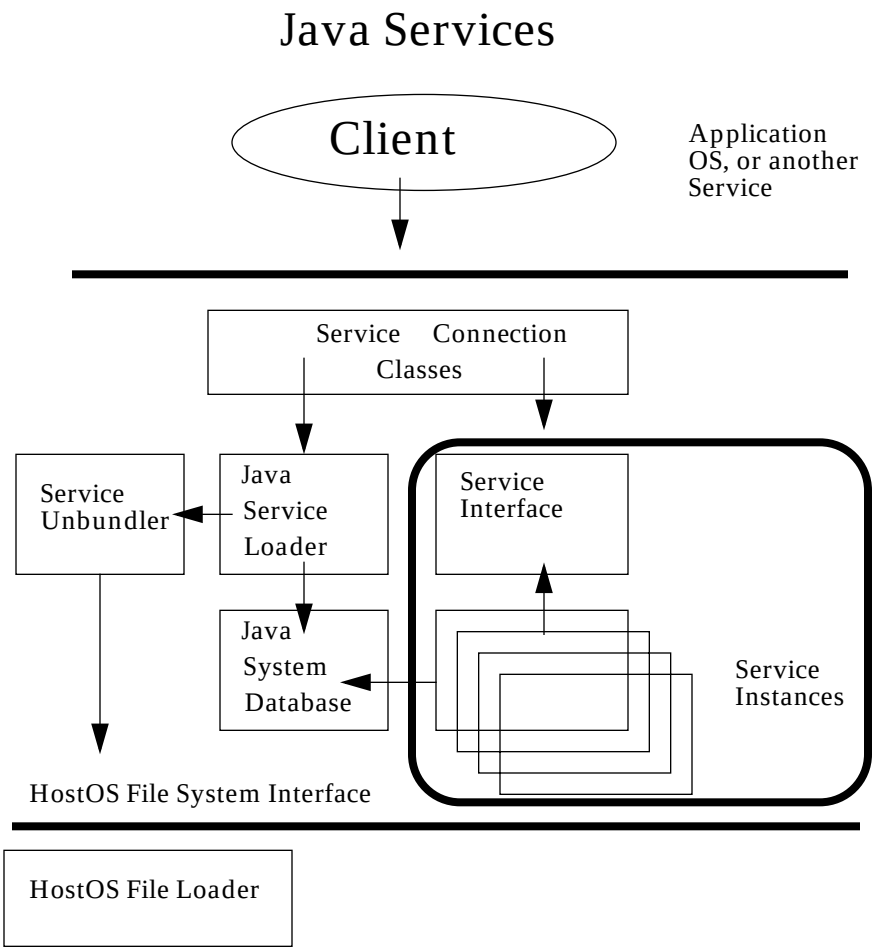


FIGURE 6-2 Java Services Architecture

JSL Classes

The JSL is comprised of a set of component classes:

- Service Connection classes
- Service Loader
- File Loader
- Unbundler

Together these components of the JSL are responsible for loading Java classes that were created independent of the software. These classes may be device drivers, applications, or a set of system services. The JSL allows these classes to be retrieved on an as-needed basis. It also defines how these classes, defined as a service, can advertise availability to other components.

The JSL uses the JSD in the following ways:

1. Retrieves business cards from the JSD for each set of loadable services. See a more detailed description of business cards later in this chapter.
2. Tracks the status of each loadable service.
3. Creates *interface* and *alias* advertisements that let users of a service find and instantiate it.

Service Connection Classes

The Service Connection classes provide methods that can be used by applications to discover, create, and connect to an instance of a driver that supports an advertised interface, such as a fax. Advertised interfaces are published by a business card `bundleInstanceInfo` parameter.

The service connection classes encapsulate the public interface to the JSL and to services constructed by the loader. The client of Java Services constructs a connection object and then directs that connection to find a particular service. If the service is already known, an instance is immediately returned to the connection. If need be, the JSL downloads the service from a container such as a JAR or ZIP file before creating an instance.

Service Loader

The Service Loader is the control point for managing service loading and instantiation. When the Service Loader receives a request from the Service Connection classes to connect to an instance of an advertised service, it carries out the request. When a disconnect is requested, the reference to the service instance is removed and its cached bytecodes become services. If no other references exist, the cached bytecodes may be cleaned up by the Garbage Collector.

File Loader

The File Loader provides logical file system services. When requested by the Service Loader, the File Loader retrieves the set of classes comprising a service. The unbundled classes are placed in a logical file system where they can be accessed via normal file system operations.

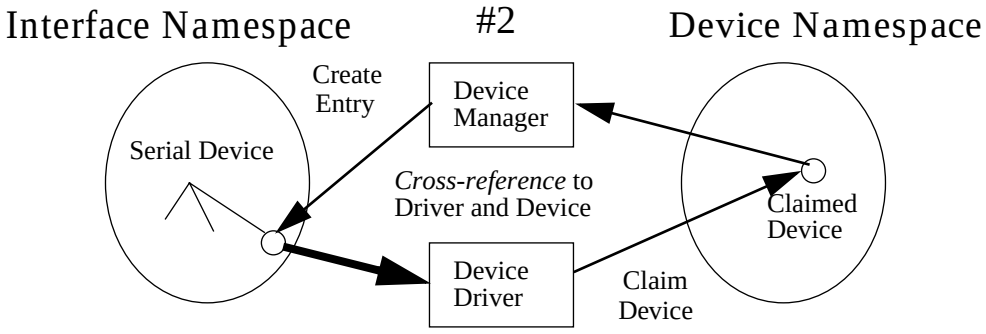
Unbundler

The Unbundler provides unbundling services for the File Loader. Support for ZIP and unsigned JAR files, as well as embedded JAR and ZIP files, is built in but other unbundled types may be loaded and advertised. Each unbundler requires file system services from the underlying host OS. The host OS services are used to read the container.

Once a container is unbundled, the JSL augments the class path so that the class loader can find classes belonging to this service. The JSL also allows some of the service classes to remain out on the server. A service can be divided into its core working set and then its lesser used classes. Subdividing the service in this manner minimizes client memory footprint and service load time.

JSL and the JDI

The JSL and the JDI populate the *interface* namespace with driver cross-reference entries to appropriate devices. This cross-reference assignment process is illustrated in Figure 6-3 below, where (#1) a device driver implements an interface and (#2) the device manager assigns a device to the driver in the *device* namespace.




```
class DeviceDriver implements SerialDevice {
.....
}
```

FIGURE 6-3 Interface Namespace Assignment

The Service Business Card

Each service is described by a class (within the service container) called a *business card*. The business card serves the following purposes:

- Names the service
- Provides vendor and versioning information
- Advertises Java programming interfaces implemented by the service
- References the service bundle containing the code to load
- Provides configuration parameters to the service

Business card entries are stored in the configuration namespace on the server-side JSD. When a client is activated, all applicable service business cards are downloaded from the server into the client JSD *software* namespace.

The JSL is a JSD event listener. As business cards appear in the client, the JSL goes about the task of activating services.

The service business card class is a Java Bean containing the following information in the form of properties. Each property has a Bean-style get and set method. When a service is installed in the JSD, each of these properties is converted to a JSD property associated with a business card entry.

TABLE 6-1 Business Card Properties

Property	Description
Title (String)	A descriptive string used by configuration tools only. This title is displayed to the user of the JSD configuration tool, usually the system administrator.
VendorName (URL)	Name of vendor that created the service. Services created by Sun Microsystems, Inc. reference Sun home page on the internet. (http://www.sun.com) This field is used on by tools manipulating the business card.
SourceCode (URL)	An optional URL to the source code used to build the service. This field is used by debuggers to support source-level debugging of services.
MajorVersionNumber and MinorVersionNumber (Integers)	Typical major and minor numbers used by the JSL and JCT to discriminate between multiple containers of the same service.

TABLE 6-1 Business Card Properties

Property	Description
MatchName (String)	A string used by the JSL to match a service with an entry in the database such as a device entry in the device namespace. Some services only load when matched to another entry.
CompatibleMatchNames (Array of Strings)	An array of strings providing alternative matching names to the JSL. The primary matching name is always used first. If need be, a service can provide additional names used to match a service to an entry.
ManagerName and ManagerType (Strings)	Manager service associated with this service. For example, PCI drivers are associated with the PCI bus manager. (See Chapter 2.)
LoadingControls (Booleans)Services	A set of three booleans that control the loading characteristics of this service. .
loadsWhenDiscovered	Services set to true when the code should always be loaded during business card processing.
loadsWhenMatchedToEntry	Services set to true when the code should load upon discovery of a matching JSD entry such as a device entry. (See matching name strings.)
loadsUponConnected	Setting to true causes the JSL to load code only when the service is actually being used by a client, thus reducing client memory footprint
BundleContainer (URL)	URL to a ZIP or JAR file containing the Java classes that implement the service.
BundleType (String)	A string naming the kind of container referenced by the container URL. Currently, the JSL supports ZIP and JAR service containers.
ResourceBundleURL (URL)	URL to a file containing NLS translatable text. (Optional)
BundleInitClassName (String)	The name of the class implementing the Java Service interface (java.system.service).

TABLE 6-1 Business Card Properties

Property	Description
BundlePackageName (String)	Prefix name for this package, comma delimited. This name is used by the JSL to avoid class naming collisions. It is required that the classes and data-files contained in each package descend from a unique Java package prefix.

TABLE 6-1 Business Card Properties

Property	Description
Bundle Class Path (URL)	An alternative place to search for classes used by this service. The JSL uses this class path to find classes not in the bundle container.
Bundle Instance Info (String)	Information necessary to create a service advertisement in the JSD interface and alias namespaces. Each service instance must at a minimum specify a logical instance name and the name of the class implementing the ServiceInstance interface. In addition to the logical and class names, a business card can also specify an advertised interface and an alias name. The format of a string describing a single service instance is: “logicalName, instanceName, interfaceName, aliasName”. The “:” delimiter may be used to specify a list instance information strings. For example, the string “myFax, myFaxInstanceClassName, java.device.fax, defaultFax : myAlternateFax, myFaxInstanceClassName, java.device.fax, alternateFax” defines two instances of a fax supported by this service. When the JSL processes this business card, two entries are created under */interfaces/java/device/fax* and two aliases are added to the *alias* namespace.
Bundle Property Name (String)	A unique prefix used to access data files within the container. This string is used by the JSL to store a unique file prefix in system properties. A service must retrieve this prefix from system properties using this parameter value as the properties keyword. This string property is only required if the service contains data files.

Role of the JSD in the Service Architecture

The JSD serves as a repository for registered system services. Registering a service with the database is accomplished by inserting a business card entry in the JSD*config* namespace.

Typically, business cards are added to the server JSD by a configuration tool running on that same server. As clients boot and connect to the server JSD, the set of applicable business cards are downloaded into the client-side JSD software namespace. Within each client, the JSL interacts with the software, interface, and aliases namespaces to manage business cards and service containers.

A separate specification on the JSD configuration tool is forthcoming. Please note that the sample JSD configuration tool is only a sample. An opportunity exists for many JSD tool suites. These tools just need to conform to the set of guidelines covered in the sample configuration tool specification.

JSL Entry Management

The JSL uses the JSD to track instances of services. A private tracking entry is created in the JSD *temp* namespace.

The following figure illustrates how the JSL uses the JSD to track and refer to service instances.

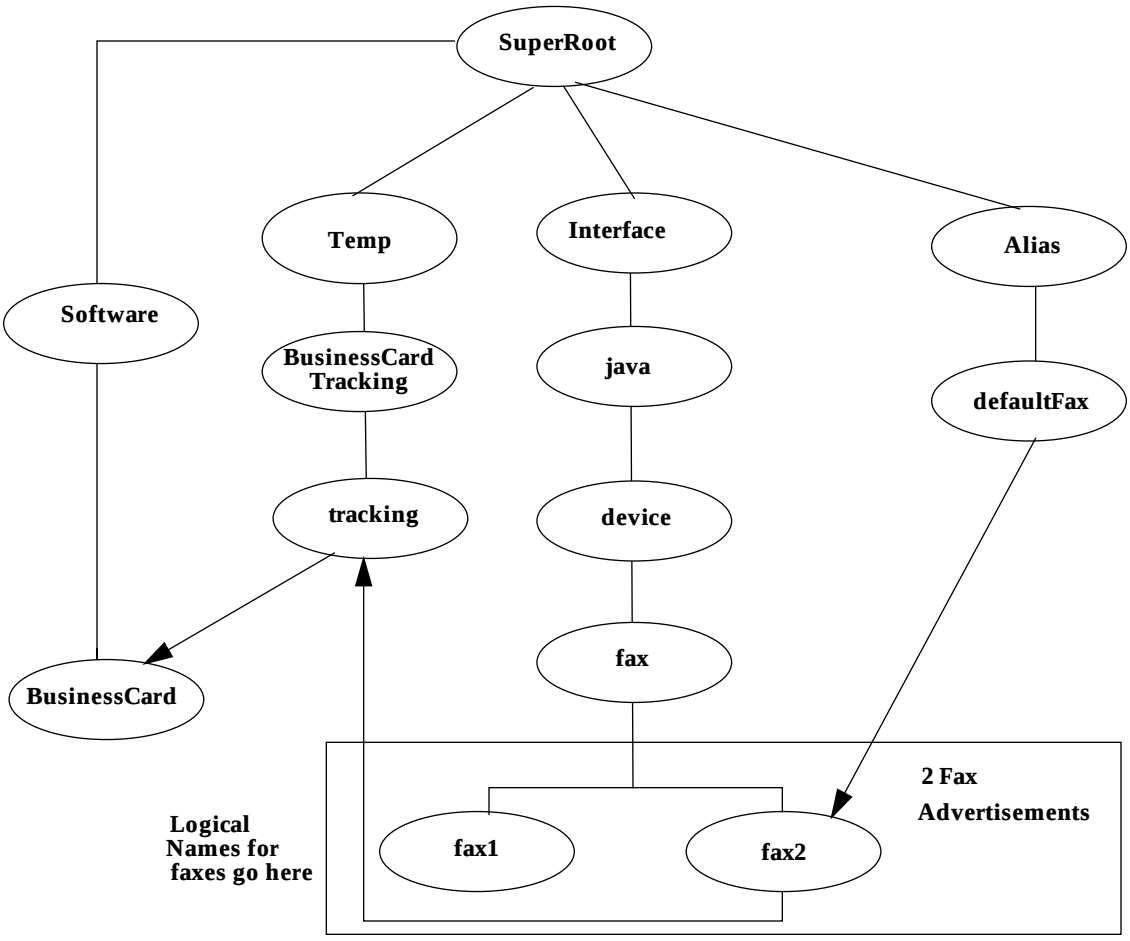


FIGURE 6-4 How the JSL uses the JSD

7

JavaOS Boot Interface (JBI)

Introduction

This chapter describes the JavaOS Boot Interface (JBI) and the booting process.

- Introduction
- Role of the JavaOS Boot Interface
- Platform Requirements
  - Standard
  - Optional
- Boot Summary
  - Microkernel
  - Virtual Machine
  - Runtime
  - Booting Process Summary
- Boot Interface
  - Start-up Interface Overview
  - Boot Operations (BootOps) Interface Overview
  - Booter and Microkernel Memory
  - Booter and Initial Microkernel Environment
- PXE (Pre-Boot Execution Environment)
- Booting Process

Information on the JavaOS Boot Interface can be found in two separate locations in this manual, in this chapter and in the *JavaOS Device Driver Guide* (JDDG). The JDDG contains native C code and technical details on the JavaOS Boot Interface, and is intended specifically for licensees who plan to customize the booter to suit a particular need or business environment.

Individuals who require planning and installation information for the JavaOS operating system, including system bootup and users and network computer management, should refer to the *JavaOS for Business Installation and Planning* document.

System and network administrators who require detailed planning, configuration, and operations instructions for the JavaOS for Business operating system, should refer to the *JavaOS for Business Network Operations* document.

Developers and OEMs requiring a customized JavaOS for Business Runtime environment should refer to the *JavaOS for Business Porting Guide.*

Figure 7-1 presents an overview of the JavaOS for Business layers, and contains the following components and interfaces:

- Client Application
- JavaOS Software Development Kit (JSDK)
- Java Virtual Machine (JVM)
- Hosting Classes
- JavaOS Device Interface (JDI)
- JavaOS Platform Interface (JPI)
- Microkernel
- JavaOS Boot Interface (JBI)

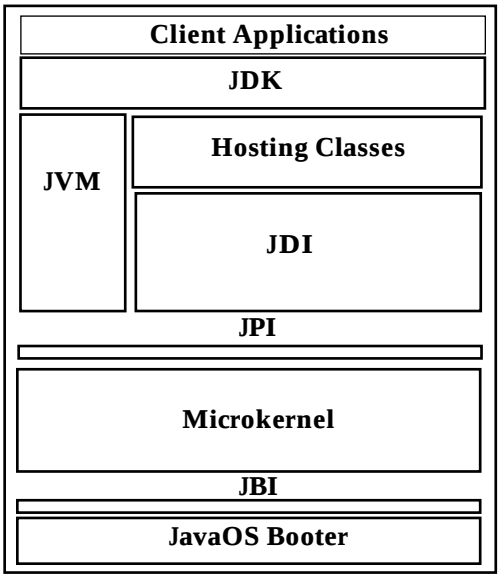


FIGURE 7-1 JavaOS for Business Layers

Role of JBI

At boot time, all computers require a mechanism from which to start the operating system. The role of the JBI is to inform the operating system of the existence of the local computing environment when a Network Computer is turned-on. Once the network computer initializes, the Boot Interface seamlessly transfers management of the local environment to the JavaOS for Business operating system.

There are two layers within the Boot Interface. The first layer, the interface itself, is the native code the operating system requires as the framework for all boot instructions. Since the Boot Interface code remains unchanged regardless of the platform it resides on, it was designed to provide a great deal of flexibility, making it excellent for custom booting configurations.

The second layer is the *Booter*. The booter is native C code that implements the interface. The booter can be tailored to suit virtually any environment, such as the use of custom PROMS for embedded devices, for example modems or phones, to the use of flash RAM/ROM for booting. Some JavaOS booter implementations may leverage an existing PROM standard, such as OpenBoot or PC-BIOS.

Note: The JavaOS Boot Interface is used to boot network computers, not to boot the server. Server booting is a separate process. Refer to the *JavaOS for Business Network Operations Manual* for a description of server initialization, booting, and configuration.

Platform Requirements

Standard

The booting system provided with the JavaOS for Business license is designed to provide flexible booting over a variety of media, including floppy disk, hard disk, and ethernet. To provide network booting of a network computer from a server, the JavaOS for Business booting system requires the following x86 configuration standard:

- Pentium-based hardware with a PCI bus
- PXE-complaint network adapter, either Ethernet or Token Ring

Complete configuration requirements can be found in the *JavaOS for Business Porting Guide*.

Optional

Boot systems for network, ROM, RAM, CD-ROM, and floppy and hard disk devices are all possible with the Boot Interface.

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 Boot Interface.

Product designers, implementation teams, and OEMs may elect to customize the boot process to suit a particular need or operating environment. To facilitate this process, the *JavaOS Device Driver Guide* (JDDG) contains complete technical booting specifications, and is accompanied by a set of C-language programs and functions.

Boot Summary

A succession of events must occur for a network computer to connect to and be recognized by the network. This process, which begins when the network computer power switch is turned-on, is performed by a series of programs designed to initialize the JavaOS platform and to bootstrap the operating system.

Collectively, these programs are referred to as the JavaOs *Booter.* In addition to the booter, a *Boot Interface* exists to pass instructions and data between the booter and other services on the JavaOS for Business platform.

The Booter is the set of C programs that initializes the booting process after power to the network computer is turned-on. The Booter initially resides on the server, until it is read into memory on the network computer. Once on the network computer, the Booter loads and starts the Microkernel.

Three portions of the operating system play an important role in the booting process:

- Microkernel
- JVM
- Runtime

Microkernel

The Booter is responsible for loading the JavaOS for Business software into executable memory and for activating the Microkernel. Once loaded, the Microkernel uses the Boot Interface to obtain information on memory usage and on Java services. In addition, the Microkernel starts the JVM.

During the booting process, the Microkernel is responsible for a number of tasks, including:

- Using the Boot Interface to find-out about memory
- Using the Boot Interface to find-out about available Java services
- Initializing the ROM file system (“in-memory” file system on disk)
- Starting the Virtual Machine
- Running `Main.java` to initialize the Java portion of Runtime

Virtual Machine

The JJVM is the target of the Microkernel. Once the Microkernel starts the JVM, the JVM starts `Main.java` to initialize the Runtime, as well as perform other functions. Some of these other functions include initializing the client network computer system database (JSD), the service loader (JSL), and the device manager.

Runtime

The *Runtime* consists of the platform-independent portions of the JavaOS for Business operating system. The point at which the Runtime takes over during the booting process is delineated by the transfer of the boot operating instruction set from C code to Java code. This transfer occurs when the Microkernel acquiesces the booter by means of `jbiQuiesce`.

During the booting process, the Runtime is responsible for a number of tasks, including:

- Initializing the JSD, JSL, and the System Events
- Calling the device manager, through the JBI, to obtain the device tree
- Initializing the JPI
- Passing memory information to the Microkernel
- Passing a Machine ID (MID) to the server

Developers and OEMs requiring a customized JavaOS for Business Runtime environment should refer to the *JavaOS for Business Porting Guide.*

Booting Process Summary

A summary of the booting process includes the following five phases:

Phase 1: Booter and Boot Interface

Phase 2: Microkernel (C code)

Phase 3: Runtime (Java code)

Phase 4: Login

Phase 5: Application

Boot Interface

The Boot Interface insulates the Microkernel from the implementation aspects of the booting system. The interface is used both by the Microkernel and the JDK Runtime layers of JavaOS for Business.

The Boot Interface defines a bi-directional interface composed of the following sub-interfaces:

- *Start-up* interface - a single function, called by the Booter, that passes control to JavaOS for Business.
- *Boot Operations* interface - an interface that allows JavaOS for Business to request services of the Booter.

Figure 7-2 depicts the relationship between these two interfaces.

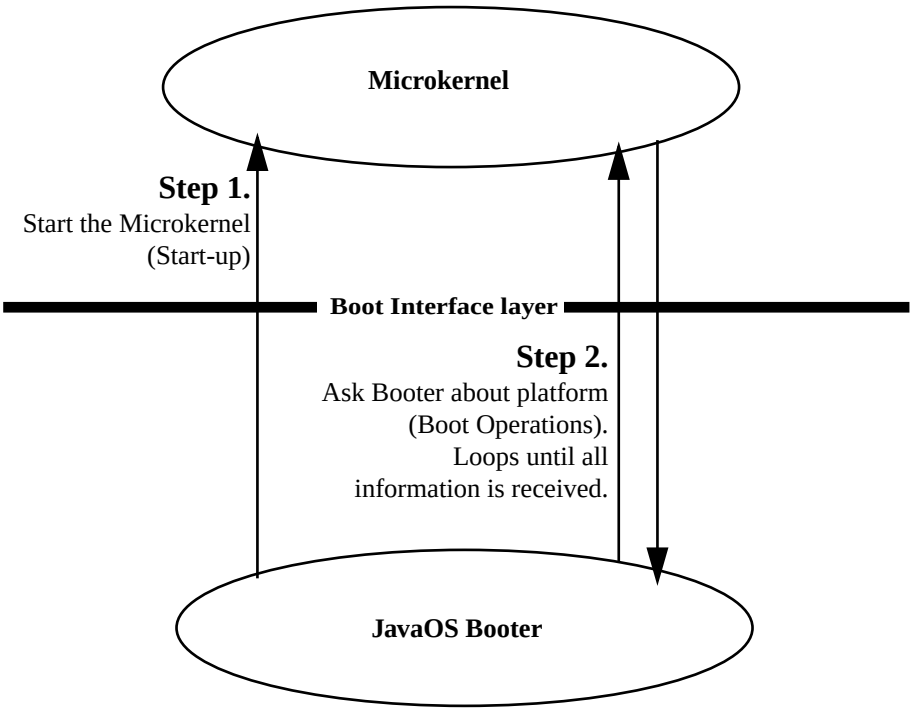


FIGURE 7-2 JBI Booter and Microkernel Interface

Start-up Interface Overview

The JavaOS for Business *Start-up Interface* consists of a single function that transfers control from the Booter to JavaOS for Business. The Start-up Interface passes parameters, including the following information, from the booting system to the operating system:

• **BootContextID**. An ID that identifies an execution context for the Booter. This ID, its value, and the represented execution context are 100% owned by the Booter.

• **BootOps Pointer.** A pointer to the structure `JBIBootOps` whose members are suitable for invocation by JavaOS for Business.

• **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.

BootContextID

Once the Booter activates the Microkernel, the Microkernel makes a series of function call-backs to the Booter. The first parameter of each call is the `BootContextID`. The typical boot scenario uses the `BootContextID` to locate the boot system global Booter data. (Global booter data both belongs to the Booter and is required to support the Booter.) The `BootContextID` is boot-implementation dependent.

Figure 7-3 illustrates this concept.

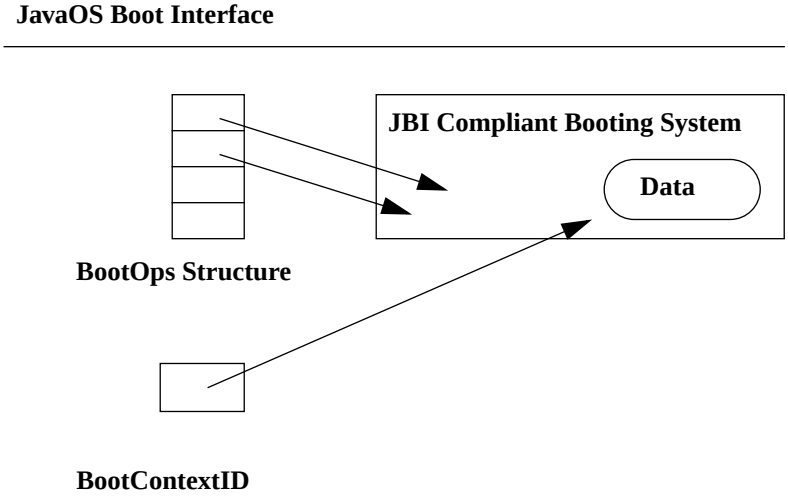


FIGURE 7-3 BootContextID Usage

Execution Environment

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

The Microkernel binary image defines the `start_JavaOS` text symbol. The Booter calls this symbol as a C subroutine passing, as parameters, the `contextID` and a pointer to the `bootOps` structure. The Microkernel then enters its booting phase, at which time it can invoke the Booter to inquire about the platform and Runtime.

Boot Operations (BootOps) Interface Overview

`BootOps` is a set of C subroutines invoked by the Microkernel during the Microkernel's booting phase.

`BootOps` functions are invoked and used by the Microkernel's current stack. The Booter must ensure that the amount of stack space used (for other function calls, local variables, and so on) is kept to a minimum to avoid stack overflow.

The clients of the `BootOps` interface are all composed of native code; primarily the Microkernel. 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 allows all clients to be abstracted from the `BootOps` structure address, its location, size, and the `BootContextID`. In addition, client invocation of boot operations can be coded as simple subroutine calls.

Figure 6-4 illustrates the call path from a native client to the boot operations implementation.

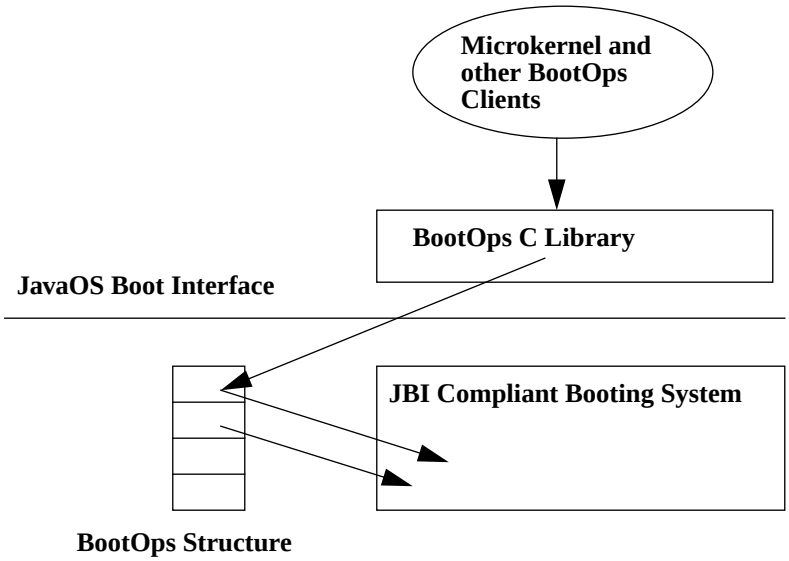


FIGURE 7-4 BootOps Invocation Flow of Control

The `BootOps` interface is designed to be transitory. That is, the complete interface implementation can be transferred from the Booter to the Microkernel. The switch is accomplished by replacing the published functions and shutting down the Booter implementation through the `BootOps jbiQuiesce` function.

BootOps Interface Service Categories

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

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

When the JavaOS Microkernel invokes a `BootOps` function, it uses the current Microkernel stack. The `BootContexID` is provided to the Booter so that it addresses the information relevant to this particular instance of the call-back.

Booter and Microkernel Memory

Physical Memory

During the booting process the Microkernel calls the Booter to obtain the map of physical memory. The map consists of a series of entries describing a range of physical memory. The possible range types are:

- Not usable
- Read-only or flash memory
- Physical RAM
- Memory for permanent I/O devices
- Memory for dynamic I/O devices
- Platform-specific control
- Downloaded File data
- Custom range name

When the Booter is ready to pass control to the Microkernel, the state of physical memory must be described to the Microkernel. This memory information includes the following:

- memory range currently occupied by the Booter
- free memory ranges
- invalid memory range or ranges
- Microkernel memory range

To accomplish this, the Booter passes the Microkernel a table describing the entire layout of physical memory.

Virtual Memory

During the booting process, the Microkernel calls the Booter to obtain the map of virtual memory. The virtual memory map communicates the current MMU configuration. The virtual memory map entry describes a range that is both virtually and physically contiguous.

When a virtually contiguous area of memory is not physically contiguous, multiple virtual memory map entries are used to describe each physically contiguous portion.

The Booter allocates the memory for the map within its own range of memory using `rangeIsBooter`. Then, a map pointer is returned to the Microkernel. The ranges in the map 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 set up in the MMU by the Booter.

If MMU is not used, the virtual memory map consists of entries that identity-map (map virtual and physical memory on a one-to-one basis) the physical memory ranges the Booter and Microkernel.

Booter Memory

Before invoking the Microkernel, the Booter allocates memory of a sufficient size to handle all boot operations. This is done to ensure that, if the Booter itself makes call-backs into other boot components (BIOS, Open Firmware, and so on), it has sufficient memory allocated for these calls. This is necessary in order to avoid memory usage conflicts between Microkernel and the Booter.

Once the Booter starts the Microkernel by invoking `start_JavaOS`, the Booter can no longer change its use of physical or virtual memory resources.

After obtaining the memory maps from the Booter, the Microkernel may begin to make use of free memory regions.

NOTE: The Booter can use the `BootContextID` as a pointer to Booter memory, for example.

Booter and Initial Microkernel Environment

In addition to the Microkernel `start_JavaOS` entry point, a number of other important expectations of both the Booter and the Microkernel are made involving the following:

- Initial Microkernel Stack
- Device Interrupt Status
- Microkernel Initialization
- File Access Information (optional)
- Support for extended functions
- Booter operations status
- Platform device information

Initial Microkernel Stack

The Booter gives 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 Instruction Set Architecture (ISA).

Primarily, the initial stack provides the task of early Microkernel bootstrap and call-backs to the Booter to obtain the physical and virtual memory maps. After this call-back, the Microkernel establishes its own memory management mechanisms and moves to another Microkernel stack. At this time, the memory associated with the initial stack can be used by the Microkernel for other purposes.

The Booter may place information on top of the initial stack (e.g., parameters to `start_JavaOS`) pertinent to the Microkernel. However, the Microkernel may modify any portion of the initial stack memory. The Booter is prohibited from placing volatile information, such as the `bootOps` structure, within the initial stack.

Device Interrupt Status

The Booter ensures that all device interrupts are disabled prior to invoking `start_JavaOS`.

Microkernel Initialization

The Microkernel must zero its own BSS (uninitialized data area). The Booter does not perform this task on behalf of the Microkernel. Only after the Microkernel acquiesces the Booter through `jbiQuiesce`, can the Microkernel reclaim any memory in the physical memory map of the content `rangeIsBooter`.

File Access Information (Optional)

A Boot Interface-compliant booter can provide the Microkernel access to an optional set of files that the Booter itself downloads. These files that can augment, enhance, or describe the required Microkernel binary image.

This feature is primarily designed to download Java classes, packages, and services such as device drivers. When the Microkernel begins to initialize the Runtime, these downloaded files append automatically to the JavaOS for Business ROM file system for easy access by the JVM class loader. The JavaOS for Business ROM file is a read-only file in memory containing classes and services needed to bootstrap the JavaOS for Business Runtime.

The set of downloaded files using the variable length data structures called the `JBIFileList`, is described in the *JavaOS Device Driver Guide (JDDG)*.

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

Single files can also be downloaded using `jbiGetFile`. In this instance, a file name and a memory address for the files passes to the Booter. The Booter transfers the file from the server to the specified memory location on the network computer.

Support for Extended Functions

In order to support additional functions that might be required for certain JavaOS implementations, the `jbiGetExtFunctions` return a pointer to the `JBIExtFunctionTable`. The JBI Extended Functions and JBI Extended Function Table, located in *JavaOS Device Driver Guide (JDDG)*, define these extended functions.

Booter Function Calls

Boot Interface `BootOps` functions include calls pertaining to file access, memory management, device tree navigation, device tree property, extended, and miscellaneous functions.

The `BootOps` interface is documented for clients in terms of the C library cover functions 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 `start_JavaOS` Microkernel entry point.

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. Refer to *JavaOS Device Driver Guide (JDDG)* for additional information.

Booter Operation Status

The Boot Interface supports an expanded error and status reporting mechanism that augments the negative (<0) return value errors pertaining to machine status and error recovery.

Platform Device Information

During the booting process, platform device information is copied by the Booter, from the JSD on the server, to the Microkernel on a client network computer. The contents of the client JSD only pertain to that particular network computer client. Information the Booter conveys to the Microkernel is an abstract data structure called a *device tree.*

The device tree contains a hierarchy and properties, which provides information about available physical devices. However, the amount of information provided by the Booter and discovered by Microkernel 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 the JavaOS for Business software.
- A combination of the above.

The actual discovery of devices can be implemented as:

- Probing and identifying hardware devices.
- Adding hard-coded data structures for fixed configuration systems.
- A combination of the above.

The Boot Interface dictates the mechanism by which information is conveyed from the Booter to the Microkernel. The physical and virtual memory maps are specifically defined, but the device tree contents are not.

The Boot Interface 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.

Figure 7-5 depicts the hierarchical nature of the device tree.

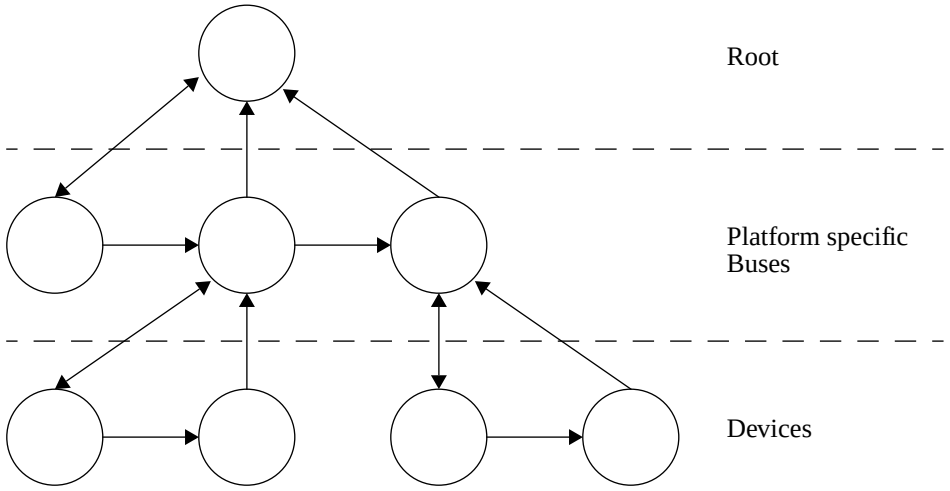


FIGURE 7-5 Device Tree Hierarchical Data Structure

The root is the top level entry of the tree. The second level of the tree represents the direct children of the root, including system level buses (PCI, SBus, ISA, and so on) or platform specific configuration entries. Devices are located at the third level of the tree, and consist of physical devices, such as SVGA cards, network adaptors, and SCSI host adaptors.

Each individual entry in the device tree contains references to its neighbor entries (refer to Figure 7-6), thus establishing a tree hierarchy. In addition, each entry refers to one or more *properties*, which are name-value pairs of entry specific information.

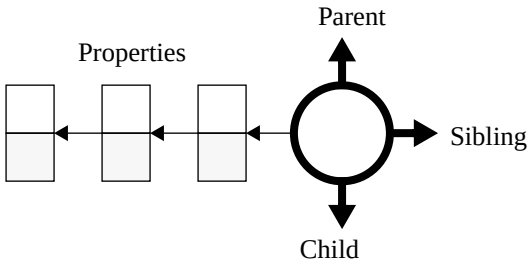


FIGURE 7-6 Device Tree References

A property is associated with each entry in the device tree. There are two property connections, a *property name* and a *property value*. The property name is a descriptive string (ASCII). The property value is an array of bytes containing information that describes a device, such as its type and address. At a minimum, each entry in the device tree must have a property name.

The JavaOS Platform Manager and Bus Manager (refer to Chapter 2, JavaOS Driver Interface) make use of the properties of various entries in the device tree. In these situations some PROMs, such as OpenBoot, can probe for devices and dynamically build a tree. The Boot Interface relies on the `OpenBoot` PROM to complete the tree building.

Booters can take other approaches, including hard-coding entries in the tree, or maintaining entries in flash memory or other local media. Device discovery is highly platform-dependent.

During different stages of the boot process, the Boot Interface examines and copies the applicable network computer portion of the JSD device tree from the server to the memory on the network computer. The I/O system may edit or append information to the device tree during this copy phase. Once the tree is copied, the I/O system begins locating bus managers and device drivers for each entry in the tree. This process is called driver matching.

Copying the device tree is final Booter step. After this occurs, the Runtime invokes `jbiQuiesce,` to stop the Booter.

Pre-Boot Execution Environment (PXE)

The Intel Preboot Execution Environment (PXE) booting standard defines a booting process for networked thin-clients (network computers). Specifically, PXE defines the network computer operating environment until an executable file containing an operating system (or OS booter program) can be downloaded from the server. JavaOS for Business uses a PXE-compliant booting system.

Following the PXE booting standard, information passes between the client network computer and the server, through the standard Dynamic Host Configuration Protocol (DHCP). This information includes an IP address assigned to the client network computer by the server, and an executable boot file sent to the client network computer by means of the Trivial File Transfer Protocol (TFTP) standard.

The executable boot file (Booter) is placed in memory on the client network computer. Once the Booter executes, it defines the remaining booting process.

PXE-compliant booting requires support from networking cards. PXE-compliant ROMs residing on an Ethernet or Token Ring card adhere to a well-defined interface understood by the Booter and the machine's BIOS.

PXE is compliant with PCI Local Bus Specifications and is a NET PC standard. For more information on the PXE booting standard, see the following website:

```
http://developer.intel.com/ial/WfM/design/pxedt/dhcp.htm
```

Booting Process

Specific booting procedures can vary greatly between networks, and even between individual network computers on the same network. However, the booting process for the default JavaOS for Business configuration can be generalized as passing through a pre-boot phase, followed by five distinct phases involving various JavaOS for Business layers.

Once power to a network computer has been supplied, the following phases are recognized:

Phase 0: Pre-Boot Environment (BIOS, PXE)

Phase 1: Booter and Boot Interface

Phase 2: Microkernel (C code)

Phase 3: Runtime (Java code)

Phase 4: Login

Phase 5: Application

An overview of what transpires between the time a network computer power switch is turned-on through the time an application loads follows, and is based upon the following assumptions:

- Network is a standard environment capable of adhering to PXE requirements.
- The server is up and running the JDK, JSD, and optionally the JCT.
- The server has been configured for the particular user and machine described.
- All device drivers for this network computer have been configured by the network administrator.
- The network computer is configured to run an application, such as the HotJava Browser.

Phase 0: Pre-Boot Environment (BIOS, PXE)

Initial network computer booting occurs at the machine level. When the client network computer is powered-up, the BIOS probes the platform hardware to locate physical devices such as the CPU, MMU, and PCI bus. Through the PCI bus, the Ethernet card and its pre-boot PXE-compliant ROM are recognized.

Through the Preboot Execution Environment (PXE) ROM, acknowledgment instructions are sent, using the Dynamic Host Configuration Protocol (DHCP), over the Ethernet to the server. The server recognizes the network computers request for acknowledgment and responds by sending an IP address back to the network computer using DHCP. This process establishes communication between the network computer and the server.

Next, the network computer requests that the server locate and send a single file (containing the Booter) to the client machine. The server responds by sending the requested file using the Trivial File Transfer Protocol (TFTP). Once received by the network computer, the Booter is started and takes over the network computer execution environment from the BIOS.

Phase 1: Booter and Boot Interface

The Booter calls the server to download all the native code in the system, including the Microkernel, JVM, and miscellaneous other C code composing the Runtime layer of JavaOS for Business.

Next, a variable set of JavaOS for Business loadable services (such as device drivers) are download one at a time using TFTP. The set of JavaOS for Business services downloaded at boot time is defined by the network administrator usimg the JCT. As the client memory fills with both native and Java code, the Booter keeps track of all memory usage, and eventually passes memory maps to the Microkernel so that the Microkernel can subsume memory management responsibilities.

After all downloads have taken place, the Booter calls the Microkernel `start_JavaOS` entry point to start the Microkernel, passing DHCP options. At this time, the Booter is ready to respond to Microkernel requests passed through the JavaOS Boot Interface (`JBI`).

Phase 2: Microkernel

The Microkernel next calls the JBI to determine the location of the loadable services in memory. Then, the Microkernel inserts each service into the ROM file system. This special file system is designed to contain all the classes and services required to bootstrap the Java-portion of the JavaOS for Business runtime layer.

Next, the Microkernel starts the JVM, which creates a Java object heap, and loads Java language bootstrapping classes (`java.lang`) from the ROM file system, in preparation for calling the Runtime start method defined in `Main.java`. Finally, the Microkernel calls the Runtime main method, passing DHCP options.

Phase 3: Runtime (Java code)

The Runtime performs a number of tasks, including initialization of the JSD, JSL, and JSE. It also starts the Platform Manager, which in turn starts the Bus Managers (ISA, PCI).

JSD activation causes the Device Manager to create a device tree using the Booter and initialize the *device namespace*. Calling the Booter Interface populates the JSD device tree. This process activates the Event System, causing the JSL to look for drivers to match with devices.

JSD Software Manager gets the Machine ID (MID) from the Booter, passes it to the server, and in response, the server sends all business cards for this particular network computer. The software manager initializes the `software namespace` with this set of machine business cards.

The system mounts any network file systems accessible via NFS, specifying the set of external volumes in a DHCP option.

If the DHCP *runtests* option is set, a set of specified self-tests runs.

Next, the Runtime initializes the window system, including the mouse and keyboard managers. (The framebuffer, mouse, and keyboard drivers required by the window system were started by the JSL in response to the set of machine business cards placed in the JSD.)

The system starts and the login manager starts, displaying a login screen soliciting a user name and password from an input devices such as a keyboard or smartcard.

Phase 4: Login

A login screen now appears. The user enters the user name and password, which is authenticated by NIS or NLS. (Protocol selection is made by the network administrator through the JCT.)

After authentication takes place, a server assigns a user ID. The login manager sends this ID back to the server, requesting all the services and application business cards specific to the user (including group business cards).

The elapsed time from Phase 0 to the appearance of the Login screen takes between ten and twenty seconds, depending on network traffic. Once the user login is successful, the Runtime starts the main application.

Phase 5: Application

The Runtime directs the JSL to load and start the Hot Java Browser.

Figure 7-7 depicts the client software status during the various booting phases.

Client Software Activity During the Boot Process

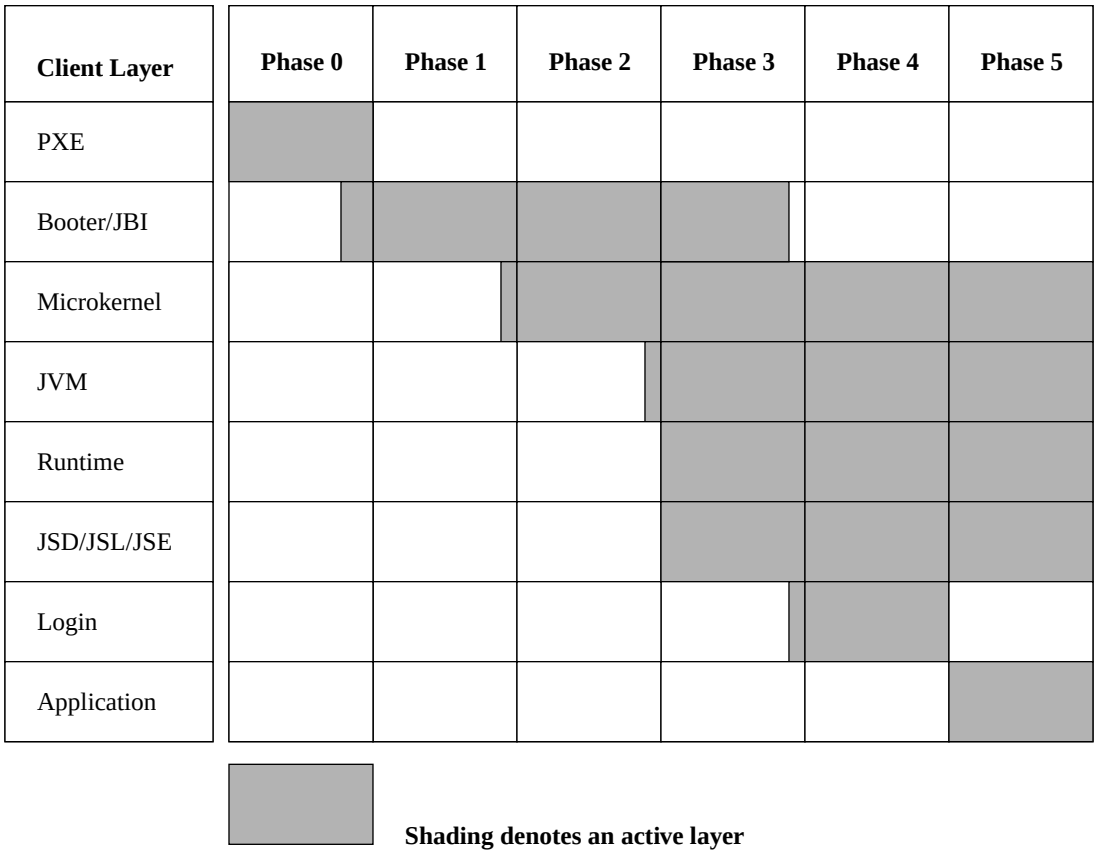


FIGURE 7-7 Client Software Activity During the Boot Proces

8

Hosting Classes

Introduction

The JavaOS for Business Runtime contains AWT and I/O support modules, generally referred to as the Hosting Classes, which use the JDI to perform I/O with a variety of devices such as mice, keyboards, and ethernet adapters.

Figure 8-1 shows an overview of the JavaOS for Business architecture layers.

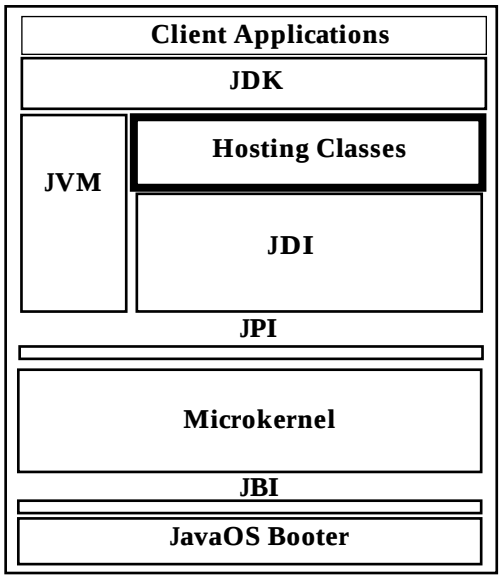


FIGURE 8-1 JavaOS for Business Layer Architecture

Figure 8-2 illustrates part of the system in more detail and includes the following components:

- Java Development Kit (JDK)
- Java Runtime layer including:
  - Java Virtual Machine (JVM)
  - Hosting Classes
  - JDI (See Chapter 2)

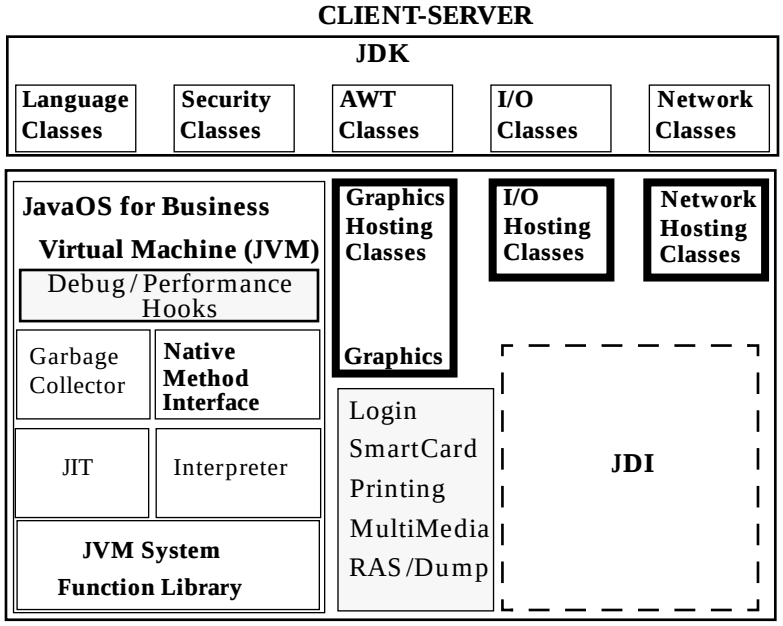


FIGURE 8-2 Hosting Classes within the Runtime Layer

The Hosting Classes in the system provide operating system support services for the JDK. As part of the upper layer of the system, they act as the gateway through which Java applications and applets access system resources.

The Hosting Classes are written completely in Java and are designed to leverage operating system capabilities to provide the best execution environment for Java programs. They do this by providing efficient access to lower level operating system services, which take advantage of the underlying platform hardware to optimize runtime execution of Java code.

The Hosting Classes are best thought of as operating system *middleware*, providing high level system services that connect Java applications and applets to the underlying operating system and platform hardware.

There are three main Hosting Classes in the system that provide support services:

- Abstract Windowing Toolkit *(*AWT) support
- Network support
- I/O support

This chapter provides an overview of the AWT and networking Host Class services. I/O support services are part of the device driver and JDI environment. (See Chapter 2, JDI for more information on JDI classes and interfaces.)

Device drivers are covered in a separate book, the JOS Device Driver Kit (JDDK), available as an adjunct to this Reference Manual.

Graphics

Working with the Graphics Hosting Classes, like the other Hosting Classes, requires that the application developer realize that the Hosting Classes function as an intermediary in reaching the underlying hardware.

The Graphics Hosting Classes describe the graphic layer that supports requirements for the Tiny AWT package. The Tiny AWT package maps low-level, platform-specific windowing components to standard Java AWT component classes in the JDK.

The low-level graphics library is responsible for all drawing to the screen. Because the library is designed to support a window system, each graphic is clipped to a specified rectilinear region.The graphics system effectively separates clip support from drawing support.

Rectilinear clip regions are stored as a list of rectangles; when drawing to a clipped region, the primitive is clipped against each rectangle in turn and the visible portions are drawn.

The actual code that controls the order in which clipped regions are applied to rectangles is highly portable and extensively used. There is little possibility, for example, of leveraging hardware for clip-region sequencing because few, if any, graphics devices support arbitrary clip regions.

On the other hand, many hardware devices support a single clip rectangle. In this case, the graphics library can be instructed to take advantage of existing hardware support. Otherwise all clipping calculations are performed in software.

Similarly, the arbitrary polygons and ellipses specified by the AWT are unsupported by most hardware. However, many graphics devices support less complex graphics primitives; the graphics library consequently implements some complex primitives (elliptical arcs and polygons) in terms of less complex primitives. These less complex library primitives can be used to leverage hardware support.

The JavaOS for Business implementation of AWT is built on top of a window system that provides the actual user-interface components and window manager services. With this system, the AWT uses a simple window system that provides basic user-interface components and window management services.

Each part of the implementation of AWT performs a different role.

- `java.awt` - AWT provides an API that is available on different Java platforms. Java programs that use AWT have similar behavior on each Java platform.

- `java.awt.peer` - The peer classes contain interfaces for each user-interface component in the AWT package. These interfaces act as wrappers for organizing how components supply their functionality to the AWT.
- `sun.awt.tiny` - The system's window manager provides window manager policy and peer user interface component classes.
- `sun.awt.aw` - The window system is the layer that is closest to owning the frame buffer. It manages graphics devices, display regions, copying screen rectangles, graphics primitives and other low-level window system functions.

The table below describes the keyboard accelerator portion of the behavior of the user-interface components provided by `sun.awt.tiny`.

Table 8-1 Keyboard Accelerator Components

Component	Key	Description
`Button`	Enter	Equivalent to a mouse press.
`Canvas`	NA	No keyboard accelerator.
`Checkbox`	Enter	Equivalent to a mouse press.
`Choice`	NA	No keyboard accelerator.
`FileDialog`	NA	Not available in JavaOS for Business.
`Label`	NA	No keyboard accelerator.
`List`	NA	No keyboard accelerator.
`Scrollbar`	NA	No keyboard accelerator.

Table 8-1 Keyboard Accelerator Components

Component	Key	Description
`ScrollPane`	NA	No keyboard accelerator.
`TextArea`	PgUp	Scroll up one page.
	PgDown	Scroll down one page.
	Home	Go to beginning of text line.
	End	Go to end of text line.
	Up Arrow	Move up one line.
	Down Arrow	Move down one line.
	Left Arrow	Move left one character.
	Right Arrow	Move right one character.
	Control-V	Paste from clipboard.
	Control-C	Copy to clipboard.
	Control-X	Cut to clipboard.
	Backspace	Delete the character before the current character.
	Delete	Delete the character before the current character
`TextField`	Home	Go to beginning of text line.
	End	Go to end of text line.
	Left Arrow	Move left one character.
	Right Arrow	Move right one character.
	Control-V	Paste from clipboard.
	Control-C	Copy to clipboard.
	Control-X	Cut to clipboard.
	Backspace	Delete the character before the current character.
	Delete	Delete the character before the current character

The `sun.awt.aw` package implements window system functionality for JavaOS for Business and provides the interface to the graphics system. The graphics interface is used to either:

- Draw window frames
- Draw within window frames

`sun.awt.aw` contains class groups in the following functional areas:

- Frame Buffer Classes
- Window Classes
- Draw Area Classes
- Cursor Class
- Event Queue Classes
- Window Region Classes
- Font Classes

The `java.awt.image` package contains classes that create and modify images. In this system, images are processed via a streaming framework that uses an image producer and an image consumer, meaning that a developer can progressively

render an image while it is in the process of being fetched or generated. Further, this framework means that an application can discard an image storage area and regenerate it as necessary. The image stream is shown in Figure 8-3.

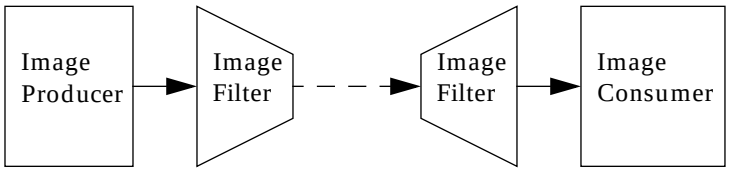


FIGURE 8-3 Image Stream

An application can insert one or more image filters between the image producer and image consumer, so that the filter can modify the image data being passed through it. The `java.awt.image` package contains a number of different image producers, filters, and consumers so that applications can be configured as needed. The application can also use the image observer, which can be used to receive notifications about an image as it is being loaded, and the color model, which specifies how to translate an image pixel value into colors.

JavaOS for Business Video

The JavaOS for Business software has been designed to make it easy to implement video applications. Developers must write an application-specific video driver (an implementation of the `sun.javaos.FrameBuffer` interface) and update the pixel building code to reflect the format required by the framebuffer (which is sometimes referred to as a *graphics card*). Adding graphics acceleration will probably improve performance, especially in critical areas. Both the video driver and the graphics acceleration typically involve a mixture of Java and native methods.

Communication with the monitor via DDC can be implemented and monitor power management can also be added.

The video driver is responsible for memory mapping the framebuffer; for initializing the framebuffer so that it can communicate properly with the monitor; for setting the framebuffer up for a known pixel depth and resolution; and for ensuring that the framebuffer is interpreting pixel values correctly.

An overview illustration of the components of the video system is given in Figure 8-4:

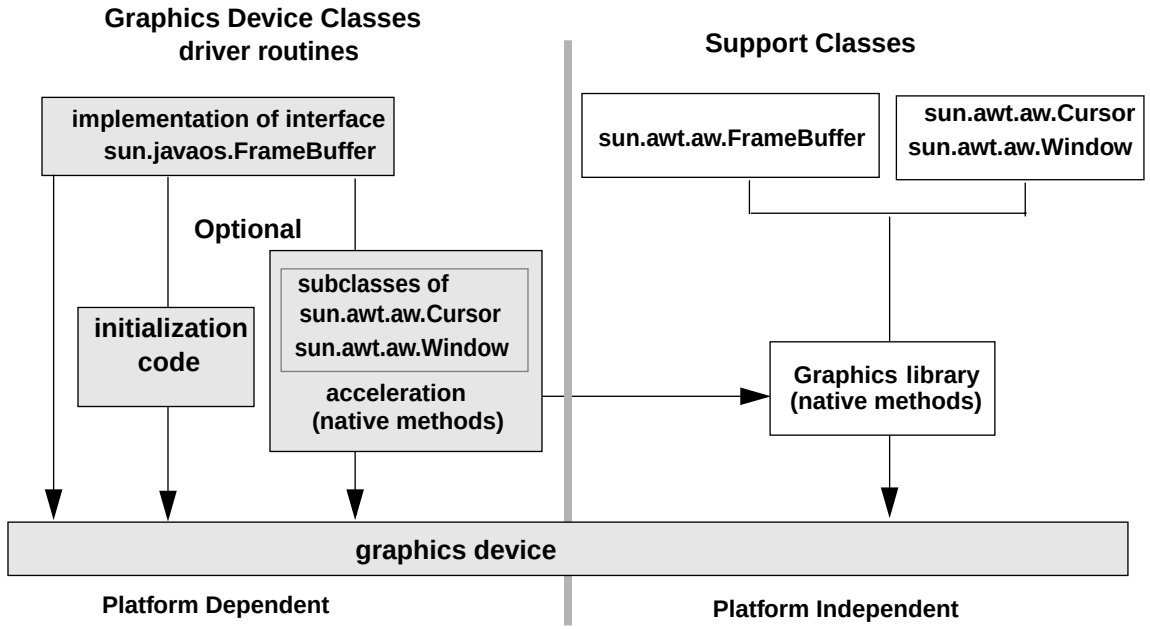


FIGURE 8-4 Overview of JavaOS for Business Video

Once this configuration is complete, the window and graphics system can interact directly with the framebuffer to draw on the screen, or the graphics system can call through the acceleration interface to use special purpose routines written especially for the framebuffer. More extensive customization and acceleration can be performed by subclassing the Window and Cursor objects and plugging in extended classes to the window system.

The actual Window and Cursor objects used by a platform should always be subclasses of the Window or Cursor classes provided by JavaSoft.

The current video framework of the software does not address Direct Graphics Access (DGA). DGA is a bypass of the high-level windowing system, often used for 3D and video applications. JavaSoft may add support for DGA when support is added for the base-level JavaMedia APIs. The software does not support multiple visuals, multiple colormaps, or transparent overlays.

Video Initialization

The video driver is responsible for ensuring that the frame buffer and the attached display are operating harmoniously. Both must operate in the same video mode and refresh rate. For example, both the frame buffer and the display could operate at a refresh rate of 60 hertz and a video resolution of 640 by 480 pixels.

The memory used by the frame buffer for displayed information is called the video RAM. Writing bytes directly to the video RAM causes pixels to light up on the display. The refresh rate is the rate at which the frame buffer scans through video RAM and outputs the contents to the digital to analog converter that drives the attached display.

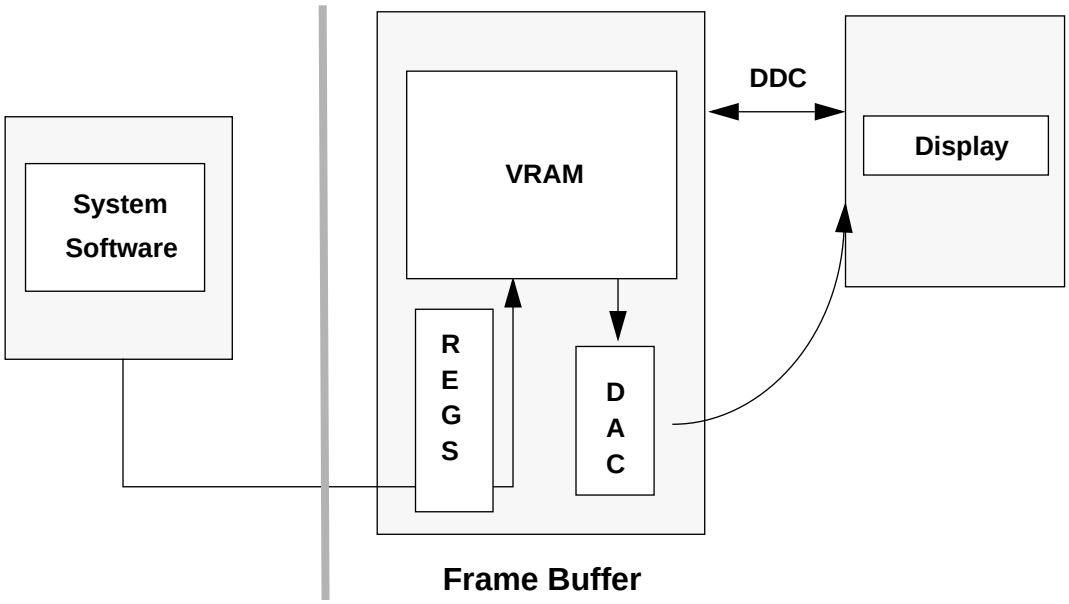


FIGURE 8-5 Overview of Low-level Video Architecture

Figure 8-5 shows the general architecture of the video system at a low level. The frame buffer synchronizes video mode and refresh rate with the attached display using Display Data Channel (DDC), a VESA consortium standard for synchronizing frame buffers with displays. The software provides support classes for enabling DDC. Using the support classes, it is possible to enable DDC by identifying the frame buffer registers for enabling DDC.

If the frame buffer does not support DDC, then the video driver can supply a list of video modes that the frame buffer supports and allow the user to pick one that is compatible with the display. Another option is to boot the video hardware in a mode that most displays support; for example, 640 x 480 at a 60 Hz refresh rate.

Once the frame buffer and display agree on a set of supported video modes, the video driver returns those modes to the software. The system requires that the frame buffer maintain a linear memory map.

In summary, video initialization involves:

- Placing the frame buffer in a known state from which it can perform DDC and memory mapping.
- Memory mapping video RAM to a range of system memory.
- Performing DDC or some other form of frame buffer / display synchronization.

During initialization, the video driver identifies the memory mapped address of pixel (0, 0) of video RAM. The native methods for accessing the frame buffer use this identified address to draw into the video RAM. The video driver must also identify how wide a pixel is in bits so that the graphics methods can properly increment through the video RAM. The video driver does not perform any I/O; it simply initializes and configures the frame buffer, and then gets out of the way.

The graphics primitives, which are not part of the video driver, are responsible for reading and writing to the video RAM.

Video Configuration

After initializing the video hardware, the next step is configuring the system colors and graphics acceleration routines.

If the framebuffer depth is 8 bits, and palette color is used, the system color table must be written into the video hardware palette. Most framebuffers require that this be done whenever the video mode is set; some, however, retain the contents of the color palette until the whole card is reinitialized.

The system color table is contained in the following C variables:

```
extern const unsigned char javaos_cmap_red[256];
extern const unsigned char javaos_cmap_green[256];
extern const unsigned char javaos_cmap_blue[256];
```

A developer will need to write a native method that loads these values into the palette registers of the framebuffer, and at the appropriate point–either in the `framebuffer setVideoMode` method or in the `framebuffer init` method–call that native method.

Configuration of the hardware palette may be the only video operation that absolutely requires native code. All other graphics operations, including register reads and writes and memory mapped I/O operations, can be performed in Java by way of the JavaOS Platform Interface. Native routines, however, may be required for performance reasons.

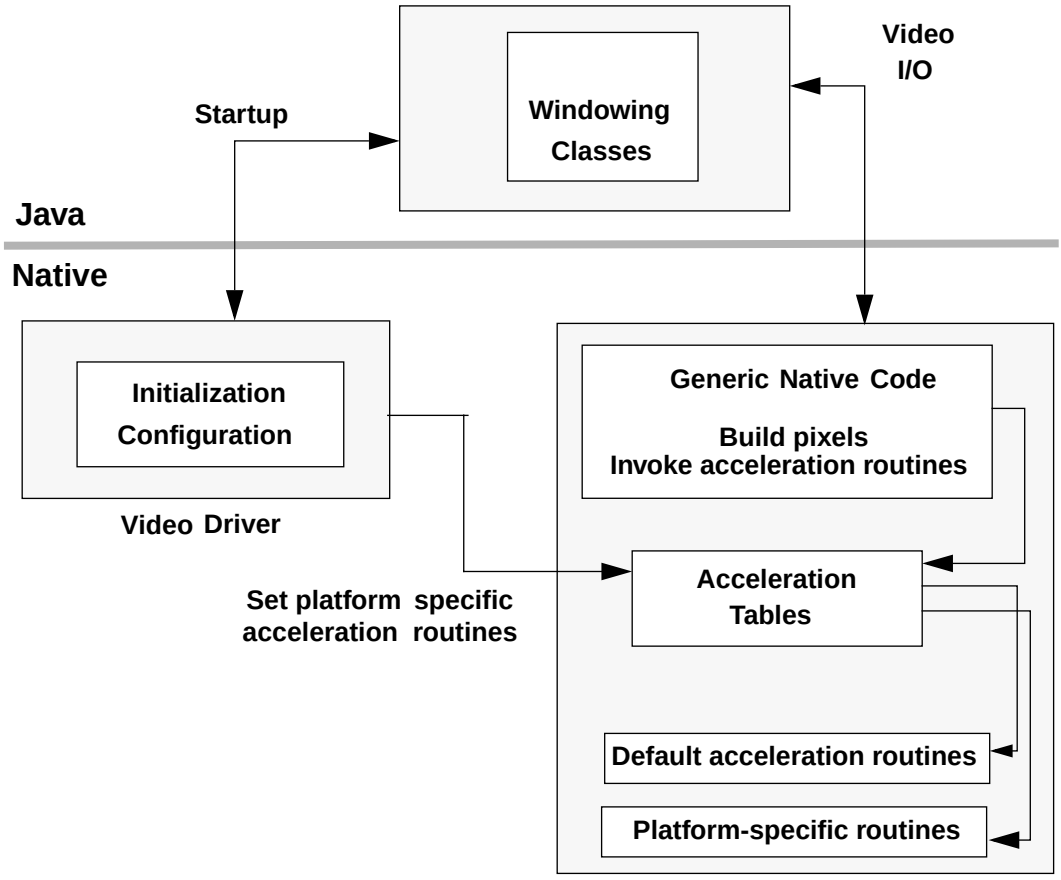


FIGURE 8-6 Detail View of Low Level Video Architecture

Figure 8-6 shows the low-level video architecture in more detail. At start-up, the video driver is invoked to perform hardware initialization and configuration. Configuration includes identifying the platform-specific native entry points for graphics acceleration in the Acceleration Table. Once the video hardware is initialized and configured, the Java graphics classes interact with standard native entry points to build platform-specific pixels from RGB values. Once the pixels are built, the platform-specific acceleration routines are invoked by way of the Acceleration Table to carry out the requested operations.

Platform-specific acceleration functions may be provided for the following operations. When no platform-specific routine is specified, the standard JavaOS for Business platform-independent routines will be invoked.

- Flat Fill Rectangle - Fills a rectangle in the frame buffer with a single color. This is useful for drawing widow backgrounds and the title bars.
- Rectangle Blit - Fills a rectangular area of the frame buffer with pixel values from another rectangular area of the frame buffer. The source and destination rectangles may overlap. Rectangle Blit is most useful for scrolling windows and moving windows around on the display.

Flat Fill and Rectangle Blit are the most important graphics primitives to optimize. Optimizations here cause the windowing system to run a great deal faster. The following additional routines may result in further performance boosts.

- Vector Line Draw - Given points x1, y1, x2, y2, connect the points with a rasterized line of pixels. There are actually two versions of this routine, one for non-vertical lines and one for vertical lines.
- Text
- Memory Blit - Move a rectangle of bits from system memory into video RAM. Useful for displaying images from files.
- XOR Rectangle - This is used for moving windows. When a user grabs a window to move it, an XOR rectangle is displayed as the window is dragged. The window fills when the user drops it. The XOR rectangle is sometimes called a feedback rectangle.

Generic versions of the primitives described above are provided by the software for all platforms using memory-mapped frame buffers. The default implementations may be overridden to improve performance on specific platforms, by overwriting the entry point values in the Acceleration Table. If a platform does not provide its own graphics primitives, then the default primitives are used. These primitives simply read and write to video RAM, without hardware optimizations. Pointers to the default routines occupy the Acceleration Table unless the video initialization routine replaces them with pointers to platform-specific routines.

To add acceleration routines, the developer must write and call a native method. Typically this method should be called whenever the video mode is changed.

Input/Output

The `java.io` package contains classes that build data streams. A *data stream* is either an input stream (which reads values from a data source, such as a Java string), or an output stream (for writing values to a data container, such as a file).

A file provides a method that returns an input stream for reading its contents or an output stream for storing values. Individual streams can be concatenated to form a chain of streams to make data flow efficient, then processed on an individual stream basis.

Because I/O is so often a function of the underlying hardware, the specifics of I/O have been assigned to the JDI and associated JPI and JSD. See Chapters 2, 3, and 4 for a more extensive discussion of how these parts of the system are related.

Network

JavaOS for Business was specifically designed to enable network devices to load and run Java programs on a network. Standard network transmission protocols such as TCP/IP and UDP are supported. Network drivers written entirely in Java transmit information between protocol stacks and network devices. Packets, frames and messages are supported.

DNS and NIS are used to look up hostnames and to supply user names and passwords. Because JavaOS for Business supports both Reverse ARP (ARP is short for Address Resolution Protocol; RARP is Reverse Address Resolution Protocol) and DHCP for discovering the network address of a machine, per-machine installation and administration costs are minimized. In addition, machines running this system can be clients of a Network File Server and managed remotely using SNMP (Simple Network Management Protocol). Moreover, these machines can obtain resources—such as the time of day— from a network server, further simplifying installation and administration. Finally, the system provides the support for sockets required by Java net classes.

The `java.net` package contains classes used in implementing network applications. In the broadest sense, the `java.net` package establishes communication links among clients, servers, URLs, sockets, packets, frames and other network resources.

- The `InetAddress` class resolves a host name to an Internet address.
- The `sockets` class communicates with Internet servers, and implements an Internet server.
- The `URL` class lets an application use URLs to retrieve Internet data, either as a complete object or as a stream.

Two exceptions, which are not subclasses of `RuntimeException`, are used for network applications. `MalformedURLException` and `ProtocolException` must be either caught or declared in the `throws` clause.

As a network centric operating system, the system supports a wide variety of industry protocols. These can be divided into four main groups, with some of the most well-know protocols listed.

- Protocols required at boot time
  - ARP/RARP
  - NFS

    - BOOTPARAMS
- Protocols required at init time
    - DHCP
    - ICP
- Protocols required at login time
    - NIS
    - DNS
    - TCP/IP
    - UDP
- Protocols required at usage time
    - IMAP
    - HTTP
    - RPC

In addition, JavaOS for Business supports the PPP protocol family (IPCP, LCP, CHAP, PAP) and SNMP agents.

JavaOS for Business Networking

Nearly all of the JavaOS for Business networking code is written in Java. There are only a few native methods of concern.

For example, when the software was ported from a SPARCStation to the x86 platform, no changes were required in the networking code, including the protocol stacks. This was true despite the fact that the SPARCStation uses a different endian model and memory model than does the Intel platform. Only one native method was required by the networking code (an IP checksum method).

The networking code is written purely in Java, deriving memory access and interrupts from the JPT. Of course, if different network adaptors are used on different platforms, the network driver must be replaced with a driver for the new hardware.

JavaOS for Business was written to use Ethernet networking devices. Ethernet devices do not require any performance-enhancing native method support.

However, for performance reasons, drivers for other network devices may require native method support. For example, a network driver for a PPP serial port—the serial port is an interrupt-intensive device with high throughput requirements, and native methods are currently the best way to optimize performance in those situations.

Figure 8-7 shows the JavaOS for Business network architecture.

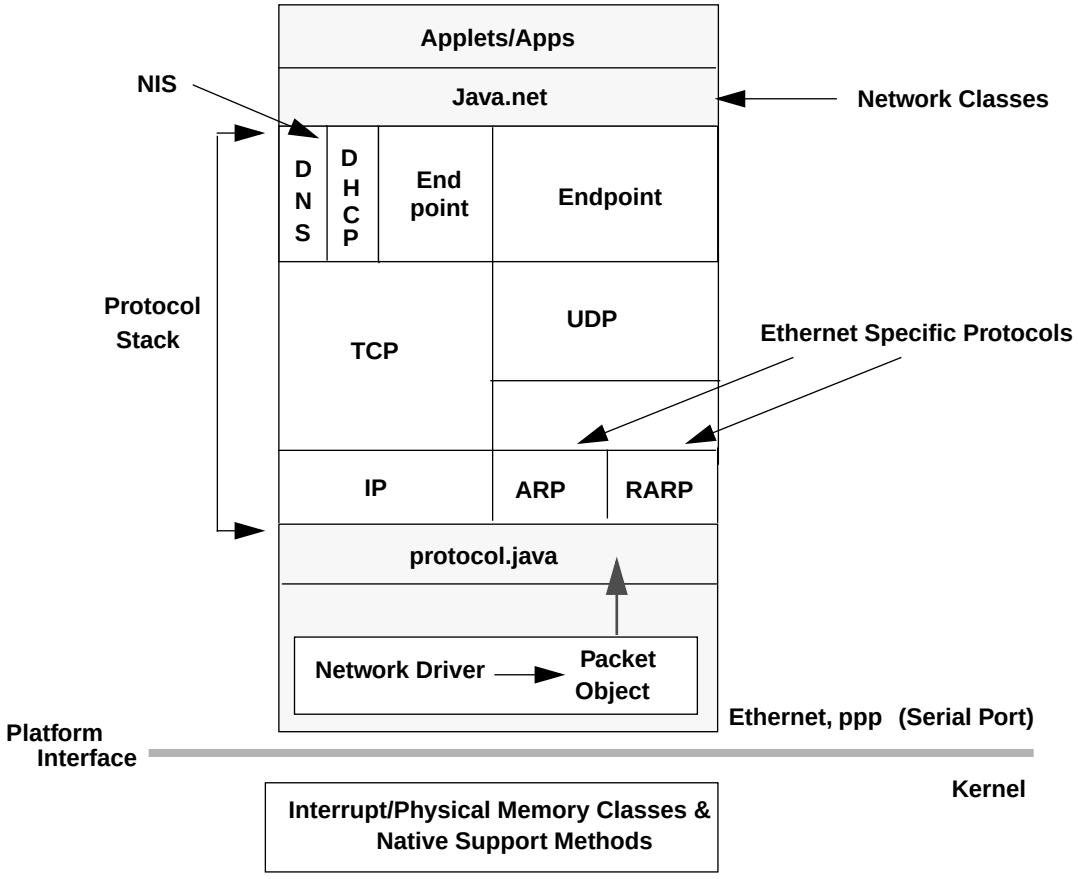


FIGURE 8-7 JavaOS for Business Network Architecture

The protocol stack in Figure 8-7 is required to implement the `Java.net` package of network access classes. The following discussion of the network architecture is framed in terms of data received from the network. The discussion is equally applicable to data transmitted to the network, with minor modifications.

Data Packet Network Transmission

First (not shown in Figure 8-7) data is received from the network by the network hardware and an interrupt is dispatched to the network device driver. A thread within the driver waits for the interrupt (which is dispatched from the Interrupt class, discussed earlier). This driver thread for processing interrupts is called the *process input* thread (see `LanceParent.processInput`). The process input thread executes at a high priority. The process interrupt thread reads the received packet from the device. The network hardware (in one Ethernet embodiment) maintains two lists in system memory. These lists include a list of packets that were received and a list of packets to transmit, and they are accessed by the hardware using DMA.

Java Packet Object

Once the packet is received and read, the driver creates a Java packet object that does not contain the packet data itself, but merely contains a reference to the packet data location in the list. This way, a memory copy is avoided. Subsequent layers in the protocol stack process the packet by accessing this location and memory.

After receiving and forming a packet, the process input thread calls up into the protocol stack, identifying the received packet by way of the packet object. The lowest level of the protocol stack is responsible for identifying the packet type and routing it to the appropriate protocol layer. The packet could be an IP packet, an ARP packet, or an RARP packet. The code that identifies and routes the packet to the higher layers is called the packet filter. It is located within the source file `ProtocolStack.java`.

Protocol Stack

ARP and RARP protocols are used by Ethernet devices to manage IP addresses. ARP and RARP are Ethernet-specific protocols and may be absent from the protocol stack on platforms that do not employ Ethernet hardware for their networking. The type of the packet is identified in the packet header.

To simplify the discussion of network architecture, assume an IP (Internet Protocol) packet was received. Once the packet is identified, it is routed to a higher layer called `IP.java`. (Had the packet been identified as an ARP packet, it would have been routed to the `Arp.java` package. If it had been identified as an RARP packet it would have been routed to the `Rarp.java` package.) Each of these upper level packages uses a common method for receiving the packet from the lower layers called `input`.

The `input` method is invoked by the packet filter with a packet object parameter. The IP package and the other packages, ARP and RARP, perform the standard processing defined by the RPC for those protocols.

The IP layer passes the packet to the next highest layer in the protocol stack, which is either `Tcp.java` or `Udp.java`. TCP stands for Transmission Control Protocol.

TCP ensures the data will arrive at its destination in the order it was transmitted. Udp makes no such guarantee. It is a less sophisticated transport protocol. UDP merely makes a best effort attempt to deliver the data. The TCP package contains a great deal of code because its function is complicated. It is at the TCP level that the context is switched from the process input thread to the application thread.

Everything above the device driver, including `protocolStack.java`, is platform independent and device independent. No porting should be required above the driver level.

The Dynamic Host Configuration Protocol (DHCP) is used in the software to initialize network and other system level parameters at boot time. These parameters include such things as the local IP address, DNS server and DNS domain name. The DHCP protocol defines the format and standard of how these options are processed.

Since the protocol standard cannot envision all the possible options that end users can think of, a vendor class option was created. This vendor class allows vendors to create their own space of options within the DHCP protocol.

The vendor class is advertised by the client in the DHCP boot packet as a string. The DHCP server receives this string and then knows the client vendor class and the special list of options that it understands.

To use the vendor class option, the developer must define a vendor class string. This string is defined by default as `JavaOS.Generic` in the `PCPlatform.java` file, and should be redefined for the specific application use before deployment.

JDK Classes

Several protocols can be invoked from TCP and UDP. One of them is DNS. Above the protocol layers such as DNS are the Java JDK classes (`java.net`), and above those are applets and applications.

In JavaOS for Business, protocol stacks are not built dynamically according to the protocol used by the network device. All protocol stacks are present in the system from boot time. These protocol stacks are statically put together according to the particular network protocol that the platform provides.

9

JavaOS for Business Classes

This chapter provides information on accessing the JavaOS for Business Classes. Class information is essential reference and programming material for licensees, developers, and OEMs needing to customize JavaOS for Business applications.

The JavaOS for Business packages are used to manipulate the system database, system events, interrupts, driver interfaces, memory, and general system services. Each specific package is comprised of classes, collections of methods (behavior) and variables (data), and interfaces, groups of methods that can be implemented by one or more classes in a package.

Access to these JavaOS for Business classes enables developers to accommodate a wide range of environment-specific needs from loadable services, such as device drivers and files services, to custom login screens.

Details on the public JavaOS for Business classes are available in a separate file as part of this document, and include information on:

- *All Packages* - a listing of the available JavaOS for Business packages
- *This package* - an interface index and a class index for the selected package
- *Class Hierarchy* - a listing of the associated variables and methods for a specific class or interface
- *Index* - an alphabetical index of the JavaOS for Business fields and methods

Note: Details on to the communications classes are available at the following web site:

http://developer.java.sun.com/developer/earlyAccess/communications.html

Access to this URL requires Java Developer Connection (JDC) registration and membership, which can be completed online and is free.

Glossary

A

ACID

See *Atomicity, Consistency, Isolation, Durability.*

Actions

Each interface in the JDI hierarchy defines its particular actions using an array of strings called actions.

Alias Entry

A JSD entry used to reference other entries in the database, including other aliases.

Applet

A program written in Java to run within a Java-compatible web browser, such as HotJava or Netscape Navigator.

ARP

Address Resolution Protocol, a type of Ethernet-specific network protocol.

Asynchronous Threading

Threading scenario in which the original driver thread is passed to a producer and is then replaced with a thread from the consumer queue.

Atomicity

JSD transaction property requiring either all state changes associated with the transaction must occur, or no state changes associated with the transaction can occur.

AWT

Abstract Windowing Toolkit, part of the standard Java interface. See also *Tiny* AWT.

B

Base Device Interface

Defines a set of generic methods that all devices support. Allows clients to get the device name or to retrieve other descriptive pieces of information such as lifecycle state, and which driver if any, is currently managing the device

Big-endian

An addressing scheme that orders bytes from the integer's least significant byte. See *Little-endian*.

Boot Interface

The interface software that loads the system into executable memory and activates the Microkernel.

Boot Operations

One of two sub-interfaces in the JBI. Allows the OS to request services of the booting system. See also *Start-up*.

Boot Operations Interface

A set of C language subroutines invoked by the Micokernel during the Microkernel's booting phase.

Booter

A program that initializes the OS platform.

Booting Layer

A layer in the OS that includes the JavaOS Booter and the JavaOS Boot Interface. The booting system loads the system into memory and begins execution of the Microkernel.

Broker

See *Event System Broker.*

Bus Management

Ensures the driver is matched to the device for each bus connected to a platform. The Bus Manager allocates memory and process interrupts on behalf of the drivers alerting the JDI post-boot to the existence of a new device.

Business Card

A JSD entry used to disseminate information to system services; includes the following information: name of service, vendor and version information, interface implemented by service or driver, reference to location of code to load, and confirmation parameters. Business cards are built from the information provided dynamically by the creator of the service or by network administration.

C

`Class` class

A special class allowing programs to inspect Java data types at *JDK Runtime*. A `class` file contains JVM instructions (or bytecodes) and a symbol table, as well as other information.

Commit

When the events in the JSD transaction event queue are posted (in the order they were received) to all interested consumers, and all locks held by the transaction are freed.

Competitive Event Consumer

Event System consumer defined as in competition for the right to completely consume an event.

Configuration Manager

Configuration Manager (CM) is the first class of Java code to be executed by the JVM invoking the CM main method with the standard Java application start-up parameters. The CM parses start-up arguments, starts up the platform manager component by passing argument strings, and starts up the list of required services and the list of option services described by the start-up parameters.

Consistency

JSD transaction property requiring that a committed transaction must correctly transform its state.

Consumer

The name given to objects that receive events.

Consumer Rule Sets

Three sets of rules for event consumers pertaining to consumption, ordering, and matching.

Consumer Queue

A queue linked to the producer master list that holds the type of event class for consumers; one queue exists for each type of event class.

Consumption Rules

Three rules for event consumers pertaining to shared event consumers, competitive event consumers, and exclusive event consumers.

CORBA

Protocol for ORB interoperability layered on TCP/IP to transfer requests between ORBs.

D

Data stream

Either an input stream (which reads values from a data source such a a Java string) or an output stream (which writes values to a data container, such as a file).

Direct Graphics Access, (DGA)

A bypass of the high-level windowing system, often used for 3D and video applications.

DDC

Display Data Channel, a standard for synchronizing framebuffers with displays.

Deadlock

When a thread times-out while waiting for read/write access to an entry lock.

Device Drivers

Code that allows the I/O to communicate with peripheral devices. Device families and stand-alone device drivers are the building blocks of the I/O system. It is the collective set of families, drivers, and their relationships that implement the I/O interfaces exported to device clients.

Device Discovery

A platform, bus, and device-specific task.

Device Exceptions

Class or interface files that execute upon error conditions. The JDI reports error conditions using *exceptions*. Device exceptions are thrown by drivers and handles and contain a reference to the handle in use when the exception occurred.

Device Handle

An application constructs a *device handle*. The device handle references the device driver (object) and a set of related action(s) or set of actions the device must perform. The system implements the device handle addressing the device driver, which in turn acts upon the specific device based on the information contained in the JSD.

Device Manager

The central logic for the JDI driver architecture.

Device Matching

An I/O process to determine the managers and device drivers in every entry in the JBI device tree.

Device Tree

A data structure that the JBI system creates to provide information about physical devices available to JavaOS; contains three levels: root, platform specific/buses, and devices.

DGA

Direct Graphics Access, a bypass of the high-level windowing system, often used for 3D and video applications.

DHCP

Dynamic Host Configuration Protocol, a type of network protocol used to initialize network and other system-level parameters at boot time.

Disconnection

The act of severing the connection of a JSD entry or subtree from either a drafted or published parent, or a child. For a child, disconnection severs all descendants of the child.

Durability

JSD transaction property requiring that, after the successful completion of a transaction, any state change must survive subsequent failures, such as a system crash.

Dynamic Host Configuration Protocol (DHCP)

A type of network protocol used to initialize network and other system-level parameters at boot time.

E

Entry

A JSD subtree; entries can be inserted, disconnected, or removed.

Event

A typed (primitive or reference) piece of information, as defined by a class, exchanged between software objects.

Event Handler

A method in the OSEventConsumer interface that is invoked by the event system broker in order to pass an event from a producer to a consumer.

Event System

JavaOS for Business software to manage, monitor, and update the exchange of class information between objects, such as file systems and device drivers. *Also*, the class (or static) methods and data defined in the `OSEventProducer` abstract base class.

Event System Broker

The portion of the Event System software that performs the management of all information passing between event producers and event consumers.

Exact Matching

Event System consumer defined as requiring that event matching take place only at the class level.

Exclusive Event Consumer

Event System consumer defined as the sole recipient of an event.

F

Filtered Matching

Event System matching between a consumer and an event using a filer, such a pathname string.

Framebuffer

In graphics, the interface used to handle I/O for graphics. Sometimes referred to as a *graphics card*.

Framed

The process of discovering a device by an entity.

G

Garbage Collector

Garbage Collector (GC) is an automatic utility program that cleans out various data heaps and caches and accounts for memory allocation within an object heap.

Generation

A polling mechanism to keep track of changes to database entries. An *entry generation* is a value that is updated whenever the entry is changed, while either published or drafted.

Graphics primitives

Basic graphics functions, such as *Flat Fill*, which filles a rectangle in the framebuffer with a single color, or *Rectangle Blit*, which fills a rectangular area of the frame buffer with pixel values from another rectangular area of the framebuffer.

H

Handle

See Device Handle.

Handler

See *Event Handler.*

Hosting Classes

Classes containing support modules for AWT, networking, and input/output functions.

I

Inheritance Hierarchy

The set of I/O interfaces for any one device.

Interrupt

An asynchronous event generated to request attention from a CPU.

Interrupt Source Entry

An entry in the Interrupt Source Tree representing an interrupt source; can be accessed by both Java and native code.

Interrupt Source Tree

An organizational structure, also called the *interrupts* namespace, that contains details about interrupt sources.

IP

Internet Protocol. Frequently used in conjunction with TCP.

Insertion

When a drafted JSD entry is placed under either a drafted parent entry or a published parent entry. Insertion affects the child entry.

Interface

A cross-reference of objects by type; pertains to JSD namespaces, such as device or software, and entries.

Interface Advertisement

A reference to a driver, and the driver references the device.

IIOP

Internet Inter-ORB Protocol. A CORBA standard protocol for ORB interoperability, layered on TCP/IP to transfer requests between ORBs.

ISA

Instruction Set Architecture.

Isolation

JSD transaction property requiring transactions to maintain a coherent state; multiple transactions must modify state serially, and cannot modify state simultaneously.

IST

Interrupt Source Tree or *interrupts* namespace. E-ach JSD entry within the interrupt source tree represents a device capable of requesting attention via an interrupt.

J

JADK

Java Application Development Kit (JADK); used in conjunction with the JDDK. A software application that allows licensees and vendors to develop portable device drivers and applications through the JavaOS Device Driver Interface (JDI).

JavaOS Event Management System

Provides the framework necessary to identify, map and manage device drivers, network protocols, and file systems to be loaded into the client platform.

JavaOS Graphics System

Supports all common graphics calls, including draw, fill, lines, arcs and polygons and font rendering. Supports the Java Abstract Windowing Toolkit (AWT) in a memory-efficient way.

JavaOS Network Classes

Includes industry standard networking protocols such as TCP/IP, UDP and ICMP for basic transport and routing.

JavaOS Window System

Controls all drawing to the screen, implements user interface components such as buttons, menus and scrollbars and handles management of overlapping windows. Supports the Java Abstract Windowing Toolkit (AWT) in a memory-efficient way.

Java packet object

An object which contains a reference to the packet data's location but not the packet data itself.

JBI

JavaOS Boot Interface. The software that loads the JavaOS into executable memory and activates the Microkernel. The JBI standardizes the boot operations available to the operating system. This interface is used both by the Microkernel and the JDK Runtime layers.

JCE

JavaOS Configuration Environment. A proprietary structure that provides services to the JCT Navigator, the JSD, and the JAR files, and provides a user interface for displaying Java Bean properties.

JCT

JavaOS Configuration Tool is a system application used to administer and configure software for JavaOS for Business Network Computers. Manages the hardware platform configuration.

JCT Navigator

An administrator interface that parses parameters and initializes the JCE.

JDI

JavaOS Driver Interface. The JDI is a series of classes and interfaces providing software access to locally connected I/O devices. The JDI calls upon the booting sub-system to publish devices found during system boot.

JDDK

JavaOS Device Driver development Kit. The JDDK and a Java Application Development Kit (JADK) allow licensees and vendors to develop portable device drivers and applications through the JavaOS Device Driver Interface (JDI). The JDDK contains a series of device interfaces, exceptions, handles, and driver classes that are collectively called the JDI for DDK.

JDK

JavaOS Development Kit. The JavaOS for Business layer that provides the means and the support for layers and components through the use of Language, Security, AWT, I/O, and Network classes. JDK configuration components depend upon factors such as B-JDK (Business), E-JDK (Embedded), or P-JDK (Personal).

JDK Runtime Layer

JavaOS for Business layer that includes the Java Virtual Machine (JVM), JDK Hosting Classes, JavaOS Device Interface (JDI), and JavaOS Platform Interface (JPI). This layer resides above the Microkernel and begins the platform-independent portion of the operating system. JDK Runtime layer components include code to support AWT, Networking, and file-related I/O classes.

JPI

JavaOS Platform Interface. JPI Services (JSD, JSL, and Event Management System) can be configured as a stand-alone system on a separate machine for client services and heavy access balancing. The JPI Services include the server component with configuration management and client facilities.

The JPI provides services to the JVM for threads, paging, and native code libraries. The OS extensions can use services like platform timing, interrupt and memory architecture.

JSD

JavaOS System Database. Manages configuration information that describes what devices are present, what system software services are installed, what user and group attributes have been selected, and any application-specific information required. The JSD lets operating system services, applications, and JDK components store and retrieve complex configuration information about all Java platforms.

JSD Event

Finding advertised interfaces relies on the JSD event production capability. Each time an advertisement entry is added to the interfaces namespace, an application can receive an event that contains a reference to the advertisement entry.

JSD Query

An application can use JSD Query process to cycle through a set of advertisement entries.

JSL

JavaOS Service Loader. A system application that manages the loading and unloading of system software services.

JSL Enumeration

A process. JSL provides a public static method to return the set of advertisements associated with an interface name. Given an interface name, applications use the static method defined in the ServiceLoader class to obtain an *enumerated* list of advertisements.

JVM

JavaOS Virtual Machine. JVM is an abstract computing machine with an instruction set that uses various memory areas. A layer that supports the Java bytecode interpreter loop, execution handling, memory management, threads, class loading, and bytecode verifier. JVM components are Garbage Collector, Native Method Interface, JIT (Just in Time) compiler, Interpreter, and the JVM System Function Library.

JVM System Functions

A collection of raw platform services required by the JVM and provided by the JPI.

K

L

Layer

JavaOS for Business can be thought of as organized in layers, separated by interfaces. The layer closest to the client is the Boot layer, and is separated from the rest of the software by the JBI. The next layer is the Microkernel, which contains much of the client-specific processing code, and is separated from the rest by the JPI. The next layer is the Runtime, which contains the JDI, the JVM, and the Hosting Classes. The next layer is the JDK, and the last layer is the client application.

Leaf Entry

A JSD parent entry without children.

LDAP

Lightweight Directory Access Protocol. A generic directory service that encompasses existing enterprise-wide directory service.

Little-endian

An addressing scheme that orders bytes from the integer's most significant byte. See *Big-endian*.

Lock

A JSD reader-writer mechanism that lets an entry be inspected or modified without interference.

M

Master Consumer List

A list maintained by the Event System containing subscribed event consumers.

Master Lists

Event System lists that facilitate event monitoring and matching for producers and consumers. See also *Master Consumer List* and *Master Producer List*.

Master Producer List

A list maintained by the Event System containing registered event producers.

Matching Rules

Three rules for event consumers pertaining to exact matching, sub-class matching, and filtered matching.

Microkernel

JavaOS for Business layer that supports booting, interrupt handling, multiple threads, traps, and DMA handling enabling users to run multiple applets at the same time or download information while running a Java application. The Microkernel is a small component of native code that provides services to the virtual machine and to the OS extensions.

MMU

Memory Management Unit. A hardware device that maps virtual addresses to physical memory addresses, or memory protection. Not a JavaOS for Business requirement.

N

Naming

A process whereby a device is discovered and named by the JBI-PROM, bus manager, or JSD administrator.

During the device publication process, the JDI may provide additional device name aliases as the device is added to the JSD. The naming of a device within the database also triggers a categorization process that results in clients being alerted with an Event.

Namespace

A specifically-designated tree of entries in the JSD that names objects of like kind, such as software or devices.

Namespace manager

A piece of software that controls how entries are stored and accessed within a particular namespace.

Native

A reference to code or requirements of a specific platform, as opposed to JavaOS for Business code, which is designed to work with multiple platforms.

NT Registry

A generic directory service that encompasses existing enterprise-wide directory service.

O

Ordered Consumers

Event System consumers defined as having an ordering preference when receiving event information with respect to other consumers of a similar class of event.

Ordering Rules

Two rules for event consumers pertaining to ordered consumers and unordered consumers.

P

PCI Expansion Bus

Peripheral Component Interconnect. A bus design that allows the host processor to access devices at speeds approaching that of a full native bus speed processor. Components designed for the PCI bus are PCI-specific, not processor specific. This isolating device design from processor upgrades.

Persistence

Server data that does not disappear when the device loses power.

Physical Address

A specific memory location in a computer or on a disk.

Physical Address Space

The domain of physical addresses available on the computing platform.

Platform Manager

The component responsible for initialization of platform-specific devices and other hardware resources. The Platform Manager is the CPU host-bus manager.

Platform-independent code

Code not dependent on a particular hardware or hardware operation environment. JavaOS for Business layered architecture includes this code and platform-specific code. Contains the JavaOS Device Drivers, JavaOS Network Classes and the JavaOS Window and Graphics systems.

Platform-specific code

Code dependent on a particular hardware or hardware operation environment. JavaOS for Business layered architecture includes this code and platform-independent code.
Platform-specific code is compiled to native code and is contained in the Microkernel and the JavaOS Virtual Machine.

Pointer

The device name serving as a pointer to a device during device discovery.

Producer

The name given to objects that are the source of events.

Properties

Name-value pairs of entry-specific information contained in the JBI device tree.

Q

Query

A structured way of asking the database for information. A JSD query includes the name, scope and state of search, entry of last match, and entry of current match.

R

RARP

Reverse Address Resolution Protocol, an Ethernet-specific type of network protocol.

Registration

The means by which the Event System Broker recognizes an event producer.

Registered Producer

An object that is the source of events that is recognized by the Event System Broker.

Removal

A two-step process that removes JSD entry or subtree from the database.

Replication

A persistent entry stored on only one server.

Reverse Address Resolution Protocol

(RARP) An Ethernet-specific type of network protocol.

Rule Sets

See *Consumer Rule Sets.*

Runtime Native Methods

Memory, interrupt, DMA, and miscellaneous APIs designed to support the JavaOS for Business device driver model.

S

SCSI

Small Computer Systems Interface.

Server Serialization

The process whereby client updates or requests occur one at a time.

Service Manager

A utility package of Java code that can locate, load, and start JDK Runtime services.

Shared Event Consumer

Event System consumer defined as willing to share events with other consumers.

Start-Up Interface

One of two sub-interfaces in the JBI. A single function that transfers control from the booting system to the JavaOS, thereby allowing the OS to start the Microkernel. See also *Boot Operations Interface*.

State

One out of three JSD entry characteristics: drafted, published, or deleted.

Sub-Class Matching

Event System consumer defined as requiring that event matching take place at any level, from the class level down through the sub-class level.

Subscription

The means by which the Event System Broker recognizes an event consumer.

Subscribed Consumer

An object that receives events that is recognized by the Event System Broker.

Synchronous Threading

A threading scenario in which a single driver thread passes to a producer and then to a consumer.

T

TCO

Total Cost of Ownership.

TCP

Transmission Control Protocol. Frequently used with IP.

Templates

Pre-formatted software allowing application developers to make extension construction easy. Sun provides base classes that act as templates for various kinds of drivers and managers.

Thread

A single sequential flowing control within a process; used to perform tasks that need to be isolated from other tasks. The Runtime schedules all threads in JavaOS.

Thread Management

The Microkernel exports a set of thread management services that let the virtual machine fulfill the semantics of Java threads. Each Java thread is layered upon a Microkernel thread.

The Microkernel thread model supports multiple threads within a single JavaOS for Business process.

Tiny AWT

A package of Java classes that map low-level, platform-specific windowing components to standard Java AWT component classes.

Transactions

Client entry or entries belonging to the same database branch.

Transaction factory

The software device used to instantiate a transaction object based on criteria.

Transience

Server data that disappears when the device loses power.

Tree

The entire JSD hierarchy from the root entry down through all its descendants.

U

Unordered Consumers

Event System consumer defined as having no ordering preference when receiving event information with respect to other consumers of a similar class of event.

V

Virtual Address

An abstract identifier of a memory location; this abstract must be translated into a physical address before access can occur. See also Physical Address.

Virtual Address Space

The total possible range of virtual addresses.

Virtual Machine (JVM)

Layer that supports the Java bytecode interpreter loop, execution handling, memory management, threads, class loading, and bytecode verifier.

Virtual Memory

A range of virtual addresses.

W

X

Y

Z

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W