



JavaOS™ for Business™



JavaOS™ for Business™ Version 2.0

Porting Guide

JavaOS™ for Business™ Version 2.0
Porting Guide

©Copyright 1998 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A.; IBM Corporation, Old Orchard Road, Armonk, New York 10504. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun Logo, Java, JavaOS and JavaOS for Business are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries, and are used under license by IBM. The JavaOS For Business technology is the result of a collaboration of Sun and IBM. IBM, the IBM Logo, AIX, OS/2, PowerPC, and RS/6000 are trademarks or registered trademarks of IBM Corp. in the United States and other countries, and are used under license by Sun Microsystems.

BSAFE is a registered trademark of RSA Data Security, Inc.

EtherExpress, Intel, and Pentium are trademarks or registered trademarks of Intel.

Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

The OPEN LOOK and Sun(TM) Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements. U.S. Government approval required when exporting the product. RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Govt is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

Copyright 1998 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis.; IBM Corporation, Old Orchard Road, Armonk, New York 10504. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Java, JavaOS et JavaOS for Business sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays et elles sont utilisées sous licence par IBM. La technologie JavaOS for Business est le résultat d'une collaboration entre Sun et IBM. IBM et le logo IBM sont des marques déposées d'IBM Corporation aux Etats-Unis et dans d'autres pays et elles sont utilisées sous licence par Sun Microsystems.

L'interface d'utilisation graphique OPEN LOOK et Sun(TM) a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun. L'accord du gouvernement américain est requis avant l'exportation du produit.

Contents

Chapter 1. Introduction	1
Pentium implementation	1
Chapter 2. Build environment	3
System requirements	3
Inside the IBM Adaptation Kit for JavaOS for Business	3
Configuring the build environment	5
Setup and tools directories	6
Chapter 3. Source code organization	9
Source tree on Solaris	9
Organizational themes	10
A brief tour	10
Top-level directory	10
Client directory	10
Build directory	11
Makefile directory	11
JDK source directory	12
JavaOS for Business source directory	13
Microkernel directory	14
Device driver directory	14
Device driver native method directory	14
Chapter 4. Build procedures	15
Build tool variables	15
Building JavaOS for Business	20
Building JavaOS for Business with SSL	21
Cleaning up the JavaOS for Business source tree	22
Using the JavaOS for Business image file	22
Adding files to the ROM file system	22
Using the Java linker (Jld)	23
Adding new source files	23
Adding a new platform	25
Replacing the local or remote login authenticators	26
Replacing the local authenticator	26
Replacing the remote authenticator	27
Appendix A. Tools	29
Jld - the Java class linker	29
Mclass files	29
Jld options	30
Retargeting Jld	31
Internal organization	32
Choosing a target system	33
Writing an output writer	34
Filizer and Stuffer	35
JavaDoc	35
Index	37

About this book

JavaOS™ for Business™ source code is distributed to licensees in a development kit called the IBM Adaptation Kit for JavaOS for Business. The *JavaOS for Business Porting Guide* provides an overview of the Adaptation Kit and includes background information describing the source code and build environment. The main purpose of this guide is to describe how a JavaOS for Business licensee can modify the JavaOS for Business source code to support new products.

Who should read this book

The *JavaOS for Business Porting Guide* is intended for programmers who wish to add to or modify the JavaOS for Business source code and build a customized version of JavaOS.

To use this book you should be familiar with the following:

- Operating systems
- Object-oriented programming
- Java programming
- C programming
- AIX, Solaris, and GNU software development tools
- Solaris and AIX network administration

How this book is organized

- Chapter 1, “Introduction” on page 1 provides an introduction to JavaOS for Business and describes the development kits that enable JavaOS for Business customization.
- Chapter 2, “Build environment” on page 3 describes the JavaOS for Business development environment. It includes hardware and software requirements in addition to instructions for setting up the build environment.
- Chapter 3, “Source code organization” on page 9 describes the organization of the JavaOS for Business source code.
- Chapter 4, “Build procedures” on page 15 describes the build procedures, including how to build JavaOS for Business image files and how to modify the build environment to support new device drivers and platforms.
- Appendix A, “Tools” on page 29 describes JavaOS for Business-specific tools.

Conventions and terminology used in this book

The following conventions distinguish different text elements:

plain	Window titles, folder names, icon names, and method names.
monospace	Programming examples, user input at the command line prompt or into an entry field, and system output.
bold	Command names, menu choices, push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combo-boxes, notebook tabs, entry fields, and directory paths.
<i>italics</i>	Programming keywords, variables, and attributes, titles of information units, initial use of unique terms, and emphasis.

Prerequisite and related information

Before using this information, you should be familiar with the content of the JavaOS for Business library:

Title	Audience and content
<i>JavaOS for Business Application Development Guide</i>	Programmers wishing to deploy a new desktop application taking advantage of the features added to the JavaOS for Business operating system.
<i>JavaOS for Business Device Driver Guide</i>	Programmers wishing to add a new device driver or modify an existing device driver and make that driver available to JavaOS for Business network computer users.
<i>ICAT Debugger JavaOS for Business - OS/2 Warp 4</i>	Programmers who need information for installing, getting started, and performing tasks with the Interactive Code Analysis Tool (ICAT) debugger for JavaOS for Business on an OS/2 Warp 4 system.
<i>ICAT Debugger JavaOS for Business - Windows NT</i>	Programmers who need information for installing, getting started, and performing tasks with the Interactive Code Analysis Tool (ICAT) debugger for JavaOS for Business on a Windows NT system.
<i>JavaOS for Business Keyboard Reference</i>	Administrators deploying JavaOS for Business in countries using different character sets and different keyboard layouts.
<i>JavaOS for Business Network Operations</i>	System and network administrators who need to plan for, configure, and manage the JavaOS for Business system on a day-to-day basis. This book also provides planning and installation steps for the JavaOS for Business operating system, including system bootup and user and network computer management.
<i>JavaOS for Business Reference</i>	Programmers requiring information on the classes and methods unique to JavaOS for Business.

Chapter 1. Introduction

JavaOS for Business provides a runtime specifically tuned to run Java™ applications directly on hardware platforms without requiring a host operating system. JavaOS for Business provides just enough operating system features to support the Java platform, thus allowing developers to provide the benefits of the Java platform on devices with limited hardware and software resources, such as network computers. This release of JavaOS for Business is designed to be installed on a Microsoft Windows Server 4.0 system and Pentium®-based network computers.

The system provides classes for a Java programmer to use or subclass to exploit features that are unique to JavaOS for Business. JavaOS for Business is designed to be portable and extensible according to how and where it will be deployed. To satisfy different levels of original equipment manufacturer (OEM) requirements for customizing the JavaOS for Business environment, the following development kits are provided:

JavaOS for Business Software Development Kit (JSDK)

The JavaOS for Business Software Development Kit provides the class files, documentation, samples, and tools needed to develop device drivers, downloadable system components, and applications that exploit features that are unique to JavaOS for Business.

Note: Because JavaOS for Business supports the Java API, it is a platform capable of running any 100% Pure Java application or applet. Therefore, the JSDK is not required for developing applications to run on JavaOS for Business. It is required only for developing applications that exploit the features that are unique to JavaOS for Business. The *JavaOS for Business Application Development Guide* and *JavaOS for Business Device Driver Guide* contain information describing the JavaOS for Business Software Development Kit.

IBM Adaptation Kit for JavaOS for Business

The IBM Adaptation Kit for JavaOS for Business enables OEMs to provide their own binary version of JavaOS for Business to run on their specific hardware. The Adaptation Kit consists of the JSDK, the JavaOS for Business source tree, and the build and debug tools that allow an OEM to build JavaOS for Business to suit their hardware. Programs that require native code (for example, video and serial port device drivers) need the Adaptation Kit because they have to be built and linked into the JavaOS for Business image itself.

Pentium implementation

The JavaOS for Business source release includes a sample implementation for Pentium-based hardware. As a licensee, you can use this implementation as a basis for new products.

Component	Description
CPU	Pentium-based system with PCI bus (166 MHz or greater)
Network adapter	<ul style="list-style-type: none">• IBM Etherjet with the Intel® 82558 chip• Intel EtherExpress™ PRO/100 with the Intel 82558 chip <p>Note: These adapters support the Preboot Execution Environment (PXE).</p>

Table 1 (Page 2 of 2). Pentium sample configuration requirements for network computers	
Component	Description
Video adapter	<ul style="list-style-type: none"> • Matrox Millenium MGA • S3 Trio 64 V+ • S3 Trio 64 V2/DX
Mouse	PS/2 mouse
Audio	<p>Sound Blaster 16 adapter</p> <p>Note: The following adapters are <i>not</i> supported:</p> <ul style="list-style-type: none"> • Any Plug and Play (PnP) version of Sound Blaster 16, AWE32, or AWE64 • Sound Blaster (8-bit) • Sound Blaster Pro (8-bit)
Ports	<ul style="list-style-type: none"> • 16550 UART Com Port • IEEE 1284 Parallel Port (standard parallel port mode only)
Minimum memory	<p>32-64 MB</p> <p>32 MB of memory is adequate for running many Java applets, for example running host or windows emulation through Host-On-Demand or the Citrix ICA Java client. For accessing arbitrary applets from the Internet or running multiple applets requiring large objects (such as images) simultaneously, 64 MB of memory is recommended.</p>

Chapter 2. Build environment

The build environment for building JavaOS for Business is a *split-build environment*. In the split-build environment, the tools for performing a build are split between two systems—AIX and Solaris.

System requirements

The JavaOS for Business build environment requires the following:

- AIX requirements:
 - AIX 4.2.1
 - RS/6000 with a PowerPC processor, 100 MHz or faster
 - A minimum of 64 MB memory
 - 2 GB available disk space, including 200 MB for the root directory
 - 200 MB swap space
- Solaris requirements:
 - Solaris 2.5.1
 - PC with a Pentium processor, 166 MHz or faster
 - A minimum of 96 MB memory
 - 2 GB available disk space, including 500 MB for the root directory
 - 200 MB swap space

Notes:

1. The AIX and Solaris systems must be connected with TCP/IP in the same domain.
2. Sufficient system resources (for example, number of processes) must be available during the entire build process.
3. A file extraction utility is required to extract the files provided in compressed .zip format on the Adaptation Kit CD. Both file compression *and* extraction utilities are required during the JavaOS for Business build process. Currently, you can obtain these utilities from Info-ZIP at <http://www.cdrom.com/pub/infozip/Info-Zip.html>. IBM makes no representation or warranty that a zip or unzip utility will continue to be available from Info-ZIP.

Inside the IBM Adaptation Kit for JavaOS for Business

The IBM Adaptation Kit for JavaOS for Business CD contains the following.

- **jsdk.zip** (JavaOS for Business Software Development Kit)

This kit enables programmers to develop device drivers, downloadable system services, and applications that run in a JavaOS for Business environment.
- **jossrtd.zip** or **jossrce.zip** (JavaOS for Business source tree)

The **jossrtd.zip** file contains the U.S. domestic version of the source tree and **jossrce.zip** contains the export version of the source tree. The JavaOS for Business source tree contains all of the source and class files needed

to build the JavaOS for Business client and server images. The setup files for the Solaris system are also included in `jossrc*.zip`. The following items are provided in binary format only:

- Internet InterORB Protocol Enabler for Java (JIE)
- OpenCard Framework (OCF)—The export version of OCF is included in JavaOS for Business. If you require the additional function, you can download the domestic version from the OpenCard Framework Web site at <http://www.opencard.org/OCF/1.0/nc>.

- **soltools.zip** (Solaris tools)

Contains the GNU 1.0 tools and JDK 1.1.4 for the Solaris system. These tools are required to build the client and server image.

- **aixtools.zip** (AIX tools)

Contains the Cygnus GNU tools and the setup script for the AIX system. These tools are required to build the client image in a split-build environment.

- **josbind.zip** or **josbine.zip** (JavaOS for Business retail binaries)

The `josbind.zip` file contains the U.S. domestic version and `josbine.zip` contains the export version. The components are organized in the following subdirectories:

- **/javaos/apps**—Contains the HotJava Browser 1.1.4. In this version, the HotJava Browser source code has the following modifications to adapt to the JavaOS for Business configuration environment:
 - Properties data is stored in the JavaOS System Database (JSD) instead of the Java properties file.
 - Configuration JAR files are used by the JavaOS Configuration Tool (JCT).
- **/javaos/boot**—Contains the PXE boot loader binary and the JavaOS for Business client boot image (either domestic or export). The domestic U.S. version, which is built with 128-bit encryption, is included in `josbind.zip`. The export version, which is built with 40-bit encryption, is included in `josbine.zip`.
- **/javaos/FONTS**—Contains the available downloadable fonts.
- **/javaos/josrv**—Contains the JavaOS for Business server components including the JavaOS Configuration Tool and JavaOS System Database binaries. It also includes the following troubleshooting tools:
 - Error log daemon
 - Error log viewer
 - Bldlevel
- **/javaos/REMOTE**—Contains the country locales, keyboards, and translated resources.
- **/javaos/x86/services**—Contains all downloadable system services associated with the hardware platforms supported by JavaOS for Business.

- **josdbgd.zip** or **josdbge.zip** (JavaOS for Business debug binaries)

Contains the debug version of the JavaOS for Business binaries in addition to the Interactive Code Analysis Tool (ICAT) for both OS/2 Warp 4 and Microsoft Windows NT 4.0. The `josdbgd.zip` contains the U.S. domestic version and `josdbge.zip` contains the export version.

- **ntsrvbin.zip** (JavaOS for Business Windows NT server binaries)

Contains the following software packages, which are required on a Windows NT 4.0 Server system to support JavaOS for Business on network computers:

- HotJava Browser 1.1.2
- JDK 1.0.2 and JDK 1.1.4
- IBM Network Station Manager
- PXE DHCP flash code .99B (Ethernet only)

- **pxesrc.zip** (PXE boot code and flash firmware)

JavaOS for Business uses the Preboot Execution Environment (PXE) as the network boot mechanism. The `pxesrc.zip` file contains the source to a bootstrap loader for use with PXE-compliant firmware. The bootstrap is brought across the network by the PXE firmware. After the bootstrap has control, it finds the

Master Configuration File (MCF) for the client machine and loads the appropriate boot image.

The source provided in `pxesrc.zip` can be used as a model to write boot firmware for JavaOS for Business.

Note: The source code does not include a PXE build environment. Writing and compiling your own boot firmware requires a C compiler, such as the Watcom compiler.

- **index.htm** (Roadmap to files on the CD)

Contains brief descriptions and links to all the files on the Adaptation Kit CD.

- **readme.htm**

Contains late-breaking product news or ups.

- **/pubs** (Documentation directory)

Contains the JavaOS for Business documentation in PostScript, PDF, and HTML format.

Configuring the build environment

To configure the split-build environment, setup is required on both an AIX and Solaris system. Copy the zip and unzip utilities to a directory that is specified in the system's PATH environment variable on both the AIX and Solaris systems. You must also check the date and time on both systems to ensure they are consistent in year, month, day, hour, and minute.

1. On the AIX system:

- a. Log in as root.

- b. Use the **smitt** command to create a build user name (for example, **oem**) with a unique user identification (UID) number and the following attributes:

- Primary GROUP=staff
- Group SET=staff,system
- HOME directory=**/home/userid** (for example, **/home/oem**)
- Initial PROGRAM=**/usr/bin/ksh**

- c. Set the password and password expiration.

- d. Create a directory named **/javaos** at the root of AIX and change the permissions on the **/javaos** directory to universal read/write/execute.

- e. Copy `aixtools.zip` from the CD to the **/javaos** directory and unzip the file.

- f. Change the current directory to **/javaos/r1.0/setup**.

- g. Type `./setup_aix` to set up the AIX build environment.

The setup script creates a **/home/userid/scripts/r1.0** directory that contains the required build script.

- h. Use the **smitt** command to export the **/home/userid/scripts** directory with read/write access to the Solaris host name.

2. On the Solaris system:

- a. Log in as root.

b. Run **admintool** to create a build user name (for example, **oem**) with the same UID number and primary group created on the AIX system and do the following:

- Create a home directory of **/home/userid** (for example, **/home/oem**).
- Ensure the **/home** directory is not automounted.
- Set the preferred shell to Korn shell.
- Set the password to normal. It must be identical to the password set on the AIX machine.

If the **/home** directory *is* automounted and you reboot the system, you might need to manually unmount the **/home** directory and reset your environment using the following commands:

```
umount /home
.kshrc
export HOME=/home/userid
```

c. Ensure that the NFS client is running by typing:

```
ps -ef | grep nfs
```

The following should be displayed:

```
/usr/lib/nfs/statd
/usr/lib/nfs/lockd
/usr/lib/nfs/mountd
/usr/lib/nfs/nfsd -a 16
```

If any of the daemons listed above are not running, type:

```
cd /usr/lib/nfs
./statd
./lockd
./mountd
./nfsd -a 16
```

d. Create a directory named **/javaos** at the root of Solaris.

e. Copy **jossrzd.zip** or **jossrce.zip** from the CD to the **/javaos** directory and unzip the file.

f. Copy **soltools.zip** from the CD to the **/javaos** directory and unzip the file.

g. Change the current directory to **/javaos/r1.0/setup**.

h. Type **./setup_solaris** to set up the Solaris build environment. When prompted for a user ID, enter an ID identical to the one you specified during AIX system setup.

The setup script creates a **/home/userid/scripts/r1.0** directory that contains the required build scripts and configuration files. Chapter 4, “Build procedures” on page 15 provides an overview of how to use the build scripts and configuration files to build JavaOS for Business.

Note: During setup, the client and server source code trees are created on the Solaris system only. For an overview of the source code organization, see Chapter 3, “Source code organization” on page 9.

Setup and tools directories

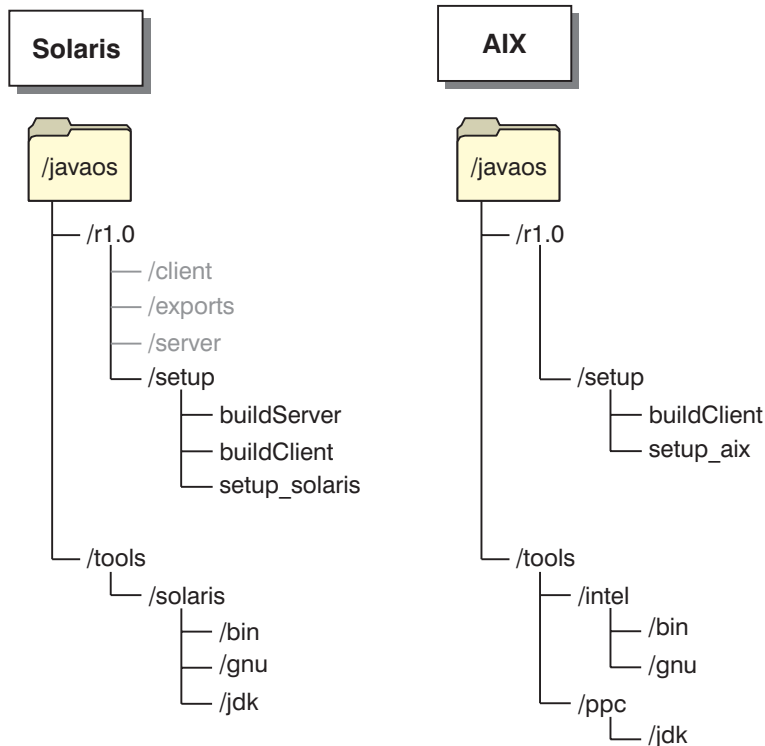


Figure 1. Setup and tools directories

Contents	Description
buildClient	Front-end script to the build system used for building the client image
buildServer	Front-end script to the build system used for building the server image
setup_*	Setup script that provides the default build configuration environment

On the Solaris system, the **/javaos/tools/solaris/gnu** directory contains the GNU 1.0 tools and **/javaos/tools/solaris/jdk** contains the JDK 1.1.4. On the AIX system, the **/javaos/tools/intel** directory contains the Cygnus GNU development tools and the **/javaos/tools/ppc** directory contains the JDK 1.1.2.

The JavaOS for Business build environment uses tools in the JDK tools directories and GNU tools directories. The build scripts automatically set up the PATH environment variable to include the directories that contain the JDK tools (javac, javadoc, and so on) and the GNU tools (gcc, gnumake, and so on).

After you run the setup scripts, the **/home/userid/scripts/r1.0** directory contains the build scripts and configuration files necessary to build JavaOS for Business. Chapter 4, “Build procedures” on page 15 describes how to modify the JavaOS for Business build files.

Chapter 3. Source code organization

The JavaOS for Business source release contains the software needed to build JavaOS for Business for different sample implementations. The JavaOS for Business source is organized to make building and modifying the operating system straightforward. This chapter describes how the JavaOS for Business source code is arranged. If you followed the setup instructions in “Configuring the build environment” on page 5, the directory structure should match what is shown in Figure 2.

Source tree on Solaris

During setup, the client and server source trees are created on the Solaris system. The JavaOS for Business source tree is part of the build tree that builds the reference platforms.

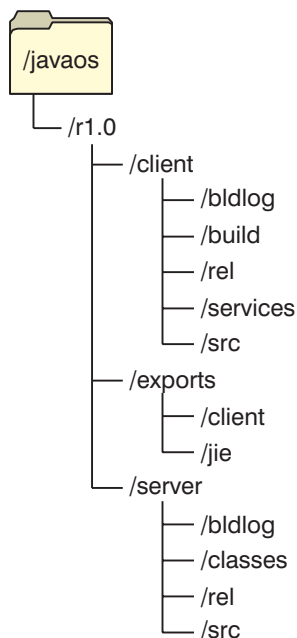


Figure 2. Source tree on Solaris

The JavaOS for Business software release is closely related to the Java Development Kit (JDK), the reference Java software release for desktop systems. Much of the source code and functionality of JavaOS for Business is borrowed from the JDK 1.1. The additions are hosting classes for supporting the JDK and a microkernel tailored to support the Java virtual machine.

There are four major levels to the JavaOS for Business code:

- *JDK Class Library*. This is a set of Java packages available on different Java platforms. The JDK class library is implemented in Java with some native methods.
- *JDK Hosting Layer*. This is a set of support classes that provide operating system support for the JDK class library. These include the window system, device drivers, and TCP/IP protocol stack. The JDK Host Layer is implemented in Java with some native methods.
- *Runtime Layer*. This includes the Java virtual machine. The Runtime Layer is implemented in C and assembly.
- *Microkernel Layer*. JavaOS for Business includes a small microkernel designed to support only the needs of the virtual machine. It is implemented in C and assembly.

Organizational themes

The JavaOS for Business source code has a few organizational themes that are useful to know about when you begin working with it or trying to find source code.

- JavaOS for Business is built in combination with modified JDK source code. Understanding the JDK source code will help a great deal in understanding how JavaOS for Business is organized. The organization of the JavaOS for Business source code roughly parallels the class hierarchy of the JDK packages.
- From an architectural point of view, JavaOS for Business is divided into two layers. Low-level code, like the microkernel code, is written in structured ANSI C and is located in the **javaos/java** hierarchy. The higher level toolkit portions of JavaOS for Business are implemented in Java and C and are located in **javaos/sun**. This Java source code uses the same organizational techniques in its directory as are used in the JDK packages.
- The microkernel code supports portability through three levels of abstraction:
 - ARCH, architecture-specific code
 - CPU, CPU-specific code
 - MACH, machine-specific code
 - common, machine-independent code (does not require modification during a port)
- Some Java classes have native methods. The source files for these native methods are not kept in the same directory as the Java class source code. (See “Device driver native method directory” on page 14.)

A brief tour

After unzipping the JavaOS for Business source code into a working directory, take a tour to see what's provided. The information included in this chapter is not meant to be comprehensive, but to capture the highlights and provide techniques for finding out more. The JDK portion of the source code is not covered in great detail because it is not an area where licensees will be working. Instead, the main focus is on the build directories and the JavaOS for Business source code directories.

Top-level directory

The top-level directory (**r1.0**) contains the following subdirectories:

Contents	Description
client	Client source tree.
exports	Location to save jossrv.zip for server from client build and location to restore JIE classes into the client tree.
server	Server source tree.

Client directory

The **client** directory contains five subdirectories:

Table 4. Client directory	
Contents	Description
bldlog	Contains the build output log.
build	Contains the makefiles necessary to build JavaOS for Business. The resulting binaries and class files are also placed here.
rel	Contains all of the built classes.
services	Contains the source code and makefiles for several JavaOS for Business downloadable system services and device drivers. The components in this directory are designed for stand-alone builds and provide samples for device driver and service developers. The resulting binaries and classes for all of the downloadable system services are also located here.
src	Main source code directory containing the source code for JavaOS for Business and a modified JDK.

Build directory

The `r1.0/client/build` directory contains the binaries and class files generated by the build procedures.

Table 5. Build directory	
Contents	Description
bin	A directory that contains one subdirectory for each built platform. The subdirectories contain the resulting binary images.
classes	Java class files common to all platforms.
classes.*	Java class files specific to each platform.
classes.tools	Java class files for the Java linker and Java Filizer.
include	Symbolic links to include files.
javaos	Build script directory.
lib.*	lib.platform contains the output of an intermediate build that can be packaged independently of the JavaOS for Business build system. This can be used to add applications to the JavaOS for Business image.
test_dir	Test suite.

Makefile directory

The `r1.0/client/build/javaos/java/java` directory contains the makefiles used to build JavaOS for Business.

Table 6. Contents of makefile directory	
Contents	Description
GNUmakefile	Lists of C and assembly sources files common to all machines.
Classfiles.gmk	Lists of Java source files common to all machines.
Classfiles-ARCH.gmk	Lists of Java source files specific to each architecture.
Classfiles-CPU.gmk	Lists of Java source files specific to each CPU.
Classfiles-MACH.gmk	Lists of Java source files specific to each machine.
Defs-ARCH.gmk	Architecture-specific macro definitions.
Defs-CPU.gmk	CPU-specific macro definitions.
Defs-MACH.gmk	Machine-specific macro definitions.
Exportedfiles.gmk	Lists of Java .class files used by javah to generate header files for native methods common to all platforms.
Exportedfiles-ARCH.gmk	Lists of Java .class files used by javah to generate header files for native methods specific to each architecture.
Exportedfiles-CPU.gmk	Lists of Java .class files used by javah to generate header files for native methods specific to each CPU.
Rules-ARCH.gmk	Architecture-specific rules.
Rules-CPU.gmk	CPU-specific rules.
Rules-PLATFORM.gmk	Platform-specific rules.

JDK source directory

The **r1.0/client/src/share** directory contains a modified version of JDK 1.1.x.

Table 7 (Page 1 of 2). JDK source directory	
Contents	Directory
doc	Miscellaneous JDK release documents.
java	Java runtime and packages such as java.awt and java.io. The cjit subdirectory contains the source code to the machine-independent parts of the just-in-time compiler. The machine-dependent parts are kept in r1.0/client/src/javaos/java/cgit.CPU .
lib	HotJava property files.
sun	Implementation code for Java packages such as java.awt and java.io.
sunw	Backward compatibility packages for JDK 1.0.2. Not used in JavaOS for Business.

Table 7 (Page 2 of 2). JDK source directory	
Contents	Directory
test	JDK test applets.

JavaOS for Business source directory

The **r1.0/client/src/javaos** directory contains the source and class files needed to build the JavaOS for Business client and server images. (The **r1.0/client/src/server** directory contains server components used to build the client.)

Table 8. JavaOS for Business subdirectories	
Contents	Description
bin	Utility scripts.
cksum	Checksum utility.
doc	Miscellaneous HotJava documentation.
filizer	Tool for preparing class and data files for inclusion with the ROM file system. See Appendix A, “Tools” on page 29 for more information on Filizer and Stuffer.
ibm	Source code for performance and serviceability.
java	Platform and CPU-specific code for the just-in-time compiler, the interpreter, the operating system startup code and the runtime. Also contains native methods for device drivers in r1.0/client/src/javaos/sun/sun/javaos and green threads.
javaos	Packages that implement replaceable components of the JavaOS for Business operating system.
javax	Package implementing client communications (Communications Port, Serial Port, and Parallel Port).
jld	The Java linker. See Appendix A, “Tools” on page 29 for more information on Jld.
lib	Property, data, and HTML files. These files contain default and startup data that is placed into the ROM file system for use at login time.
sun	The bulk of the JavaOS for Business source code organized into package directories.
sun.CPU	Not used.
sun.PLATFORM	Platform-specific portion of JavaOS for Business hierarchy and the Platform and top-level Nexus classes.
tests	Test applets and applications.

Microkernel directory

The **r1.0/client/src/javaos/java** directory contains source code to the JavaOS for Business microkernel as well as portions of the virtual machine. Table 9 on page 14 describes the subdirectories in the microkernel directory. Most of the source files in the microkernel hierarchy contain C and assembly code.

Subdirectory	Description
<i>cjit.CPU</i>	CPU-specific portions of the JIT (not supplied in source form).
<i>green_threads</i>	Green threads directory.
<i>java</i>	JDK supported classes.
<i>javai</i>	Startup routine for Java interpreter.
<i>os.ARCH</i>	Architecture-specific portions of the microkernel.
<i>os.CPU</i>	CPU-specific portions of the microkernel.
<i>os.MACH</i>	Machine-specific portions of the microkernel.
<i>os.common</i>	Machine-independent portion of the microkernel.
<i>os.devices</i>	Low-level device code.
<i>runtime</i>	Java runtime, garbage collection, and heap management.

Device driver directory

JavaOS for Business device drivers are written in Java with native methods. The **r1.0/client/src/javaos/sun/sun/javaos** directory contains Java source code for the JavaOS for Business device drivers and other parts of the JDK host layer.

Device driver native method directory

The **r1.0/client/src/javaos/sun/javaos** directory contains C source code for the native methods in JavaOS for Business device drivers and other parts of the JDK host layer.

Chapter 4. Build procedures

The JavaOS for Business build system constructs a JavaOS for Business binary for a target platform based on certain build options. The build system uses build scripts—**buildClient** and **buildServer**—to build the client and server images, respectively. The build scripts manage a small number of makefiles that perform the software builds. Keeping the number of build files to a minimum simplifies maintenance.

Build tool variables

The build scripts and makefiles are initially configured for the JavaOS for Business environment. This section shows how to modify the build configuration for a different environment.

The build scripts use configuration files for additional parameter specifications. The **buildClient** script uses the **buildClient.cnf** file and the **buildServer** script uses the **buildServer.cnf** file. The configuration files consist of a table of parameters, which are listed in Table 10. Using these parameters, you can specify a build-time configuration. (The build scripts and configuration files are located in the **/home/userid/scripts/r1.0** directory.)

Script Variable	Description	Example
buildType	Valid parameters: PC Builds the Intel client image SRV Builds the server image	PC
srcLocation	Hostname of machine where source code is located.	solserv
solarisMach	Hostname of Solaris machine where Solaris tools are located.	solserv
aixMach	Hostname of AIX machine where compiler is located.	aixserv
buildRoot	Root of source tree (based on machine type).	/javaos/r1.0/client
mountDir	Mount point on the AIX system at which the source tree on the Solaris system will be mounted.	/javaos/oem
userid	User ID to use during build process.	oem
toolsDirAix	Root of tools tree on AIX.	/javaos
toolsDirSol	Root of tools tree on Solaris.	/javaos
solarisJDK	Version of JDK used for Solaris.	v1.1.4
solarisGNU	Version of GNU compiler used for Solaris.	v1.0

Table 10 (Page 2 of 4). Build-time configuration parameters		
Script Variable	Description	Example
aixJDK	Version of JDK used for AIX.	v1.1.2
intelGNU	Version of GNU compiler for Solaris.	ibm-971219
aixGNU	Version of GNU compiler for AIX.	ibm-971219
<i>Valid targets to build (allows for customization of build-specific pieces in GNUmakefile).</i>		
solarisTargets	<p>Valid values are:</p> <ul style="list-style-type: none"> • Solaris • ServerSide • services <p>The target value can also be specified from the build command line.</p>	Solaris,ServerSide,services
aixTargets	<p>Aix</p> <p>The target value can also be specified from the build command line.</p>	Aix
buildStyle	<p>Valid parameters:</p> <ul style="list-style-type: none"> • retail • debug • perf(ormance) 	debug
buildFlags	<p>Build-time flags. The build-time flags are available for debug and performance builds only. If specifying more than one build flag, each flag must be separated by a semicolon (;) and there must not be any spaces between the flags. Specify none if you do not want to pass any build-time flags.</p> <p>The build-time flags can also be specified from the build command line. Any flags specified in the .cnf file are overridden by those that are specified directly from the command line. (See Table 12 on page 21 for descriptions of the supported flags.)</p>	IPJOS=true;CDEBUG=true;

Table 10 (Page 3 of 4). Build-time configuration parameters

Script Variable	Description	Example
buildSecurity	<p>Specifies encryption type. Valid parameters:</p> <ul style="list-style-type: none"> • export for 40-bit encryption • domestic for 128-bit encryption <p>Note: This parameter is valid only if you have installed the required BSAFE/SSL files. Contact your JavaOS for Business provider for information on obtaining the required files.</p>	domestic
getExports	<p>Valid values:</p> <ul style="list-style-type: none"> • y for yes • n for no <p>If set to yes, files are copied from the /javaos/r1.0/exports/jie directory to the client tree to build the client and jossrv.zip is copied from /javaos/r1.0/exports/client to the server tree to build the server.</p>	y
publishClient	<p>Publish exports from client build.</p> <ul style="list-style-type: none"> • y for yes • n for no 	y
publishJIE	<p>Publish exports from jie build to server.</p> <ul style="list-style-type: none"> • y for yes • n for no 	y
copyCnfFile	<p>Copy configuration file from primary environment to secondary environment.</p> <ul style="list-style-type: none"> • y for yes • n for no 	y
<i>Version control information</i>		
language	ISO 639 two-character language code.	en
country	ISO 639 two-character country code.	en
vendorShortName	Vendor short name.	IBM
vendorLongName	Vendor long name.	International_Business_Machines_Corp

Table 10 (Page 4 of 4). Build-time configuration parameters		
Script Variable	Description	Example
buildLevel	Build level string (m.n) [numeric].	0.0
buildSubLvl	Build sublevel.	0
driverName	Driver name.	g18
CMVCRRelease	Can be used for OEM-specific library information (optional).	javaos.gem
Exports		
stgServerHost	Hostname of the Solaris system. Note: Do not modify the <i>stgServerHost</i> parameter.	solserv
stgPubDir	The stage exports directory location on the Solaris system. This location is used to save jossrv.zip for the server, which is generated by the client build. This location is also used to restore the JIE classes into the client tree for building the client. Note: Do not modify the <i>stgPubDir</i> parameter.	/javaos/r1.0/exports
stgPubMnt	Mount point on the AIX system at which the <i>stgPubDir</i> directory will be mounted.	/javaos/r1s

Figure 3 on page 19 shows a sample buildClient.cnf file.

```

# buildClient
# Define the split build environment
#
# build time configuration parms
buildType      PC          #valid parms:  PC(intel)
                #          SRV
srcLocation    jose231    #hostname of mach where src is located
solarisMach    jose231    #hostname of solaris X86 machine where JDK is loc
aixMach        jose233    #hostname of AIX mach where compiler is loc
buildRoot      /javaos/r1.0/client #root of src tree
mountDir       /javaos/build #directory to mount over
userid         build      #user id to use in build process
toolsDirAix    /javaos    #root of tools tree
toolsDirSol    /javaos    #root of tools tree
solarisJDK     v1.1.4     #version of JDK used for solaris
solarisGNU     v1.0       #version of GNU Compiler used for solaris
aixJDK         v1.1.2     #version of JDK used for aix
intelGNU       ibm-971219 #version of gnu compiler for solaris
aixGNU         ibm-971219 #version of gnu compiler for aix
#
#valid targets to build
solarisTargets Solaris,ServerSide,services
aixTargets     Aix
buildStyle     retail     #valid parms: retail, debug, perf(ormance)
buildFlags     none      #build time flags (none if none to pass)
buildSecurity  domestic   #valid parms: export or domestic
getExports     y          #y(es) n(o)
publishClient  y          #publish exports from client build y(es) n(o)
publishJIE     n          #publish exports from jie build to server y(es) n(o)
copyCnfFile    y          #copy config file to other env
#
#version control infomation
language       en         #ISO 639 (2 char) language code
country        en         #ISO 639 (2 char) country code
vendorShortName IBM       #Vendor short name
vendorLongName International_Business_Machines_Corp #Vendor Long name
buildLevel     0.0        #build Level string (m.n) [numeric]
buildSubLvl    0          #build sub-level
driverName     g26        #driver name
CMVCRelease    javaos.gem #CMVC release
#-----
# Exports
stgServerHost  jose231    #hostname of stage server
stgPubDir      /javaos/r1.0/exports #location of stage exports
stgPubMnt      /javaos/r1s #mount dir to use

```

Figure 3. Sample buildClient.cnf file

Building JavaOS for Business

The JavaOS for Business binary is constructed by the buildClient script located in the `/home/userid/scripts/r1.0` directory. This shell script handles the build options and detects conflicts between selected build options and previously built modules in the build tree. It calls `gnumake` to perform the software build.

To build JavaOS for Business:

1. Log in on the Solaris system as the build user (for example, `oem`).
2. Change the current directory to the `/home/userid/scripts/r1.0` directory.
3. To build the client image, use the following command:

```
./buildClient build-options
```

where `build-options` can be any of the values listed in Table 11.

Option	Description
-clean	Builds by cleaning the tree first—removes class, object, and temporary files and then launches the build procedure. The default is to <i>not</i> clean the tree before building.
-clean_only	Cleans the build tree and exits without performing a build.
-nolog	Prevents piping of the output messages to <code>r1.0/client/bldlog/build.out</code> .
-sslget	Copies the content of <code>r1.0/exports/ssl</code> (stage area) to <code>r1.0/client/build/ssl</code> and then builds the client using the Secure Sockets Layer (SSL) files located in the <code>r1.0/client/build/ssl</code> directory. This option is valid only if you have installed the required BSAFE/SSL files. Contact your JavaOS for Business provider for more information.
-sslloc	Builds the client using the Secure Sockets Layer (SSL) files that are local in the build tree, located in the <code>r1.0/client/build/ssl</code> directory. This option is valid only if you have installed the required BSAFE/SSL files. Contact your JavaOS for Business provider for more information.
-f	Passes a build-time flag or flags. See Table 12 on page 21 for a description of the build-time flags.
-t	Passes specific target strings to make. Valid targets include ServerSide, Services, Solaris, and Aix.

Note: Some of the properties of the GNU tools require the user of the build script to have write access to the `/opt` directory in order to build on the Solaris platform.

The buildClient script acts as a front-end to the makefiles in **r1.0/client/build/javaos/java/java**. There are several options available in those makefiles that are not available as options to buildClient. The makefile options are described in Table 12 on page 21.

Table 12. Makefile options		
Option	Values	Description
CDEBUG	true false	Builds in the conditional debug facility.
CONSOLE_ENABLED	true false	Allows message I/O to the serial port.
IPJOS	true false	Builds a debug kernel.
JDEBUG	true false	Builds in the debug trace facility.
NETSHELL	true false	Builds a JavaOS for Business image that will accept telnet connections on port 20000.
NOERRLOG	true false	If false, builds in the error log facility.
NOTRACE	true false	If false, builds in the performance trace facility.

To use a makefile option with the build script, append the option to the command line using the -f parameter. For example:

```
./buildClient -clean -f CDEBUG=true JDEBUG=true -t Aix Solaris
```

Build times vary depending upon workstation model and build options. When the build finishes, the JavaOS for Business binary is placed in a platform-dependent subdirectory of **r1.0/client/build/bin**.

You must build the client and server images separately. To build the server, the client must have been built first on the same machine. Use the same procedure to build the server substituting the buildServer script and corresponding buildServer.cnf file in place of the buildClient script and buildClient.cnf file.

For example,

```
./buildServer -clean
```

A target value does not need to be specified. The buildServer.cnf file contains the required setup for building the server files. The resulting files include the configuration beans (*.jar files) located in **r1.0/server/rel/server/retails/jars** and the *.zip files located in **r1.0/server/rel/server/retail/zips** (excluding jossrv.zip, which is imported from the client build).

Building JavaOS for Business with SSL

The Secure Sockets Layer (SSL) protocol is an Internet standard for providing privacy of communication. SSL allows applications to negotiate and use cipher suites that contain different encryption algorithms, cryptographic keys and message authentication system.

SSL is commonly used for implementing HTTPS (the WWW protocol for secure HTTP), which allows Web browsers to use secure transactions over the Internet. Because JavaOS for Business provides a platform for Web

browsing, SSL provides an important security component to applets that are running on a network computer that uses JavaOS for Business. SSL can also be used by Java applets to implement other kinds of secure communications over the Internet between an applet running on an network computer and other remote software.

Building JavaOS for Business with SSL requires that you have installed the correct BSAFE/SSL files. Contact your JavaOS for Business provider for information on obtaining the required files. After installing the required BSAFE/SSL files, you can build the JavaOS for Business client image with either 128-bit encryption or 40-bit encryption. The encryption type can be specified using the *buildSecurity* parameter in the buildClient.cnf file. For 128-bit encryption, set the *buildSecurity* parameter to **domestic**. For 40-bit encryption, set the *buildSecurity* parameter to **export**.

To build the client image with SSL, specify either `-sslloc` or `-sslget` as build options. For example,

```
./buildClient -clean -sslget
```

Note: The `-sslloc` and `-sslget` build options specify where to locate the BSAFE/SSL files. See Table 11 on page 20 for more information.

Cleaning up the JavaOS for Business source tree

There are a few methods available for cleaning up the JavaOS for Business source tree. The **-cleanonly** option for the buildClient script removes class, object, and temporary files from the source tree. The **-clean** option for the buildClient or buildServer script removes class, object, and temporary files from the tree and then launches a build procedure. Also, the GNUmakefile has a clean option for removing intermediate files.

Using the JavaOS for Business image file

After successfully completing the build procedure, the JavaOS for Business build system creates a JavaOS for Business image file in the build destination directory. This directory is located in the build directory and has the same name as the build target. For example, the JavaOS for Business image file for the PC build is located in the **r1.0/client/build/bin/PC** directory. The name of the resulting image file depends upon the build style (specified by the *buildStyle* parameter) as follows:

javaos	Retail build image
javaos.debug	Debug build image
javaos.jtprof	Performance build image

A JavaOS for Business platform can boot the JavaOS for Business image file over a network. *JavaOS for Business Planning and Installation* provides instructions for booting the JavaOS for Business retail image file over a network.

ICAT Debugger JavaOS for Business - OS/2 Warp 4 and *ICAT Debugger JavaOS for Business - Windows NT* provide instructions for using the debug image file with the ICAT debugger.

Adding files to the ROM file system

When booted from a network computer, the JavaOS for Business image is downloaded from a server, or read from a Flash Card by *bootcode*. This bootcode resides in read-only memory (ROM) in the hardware. The ROM file system is a special local file system for storing permanent data and class files in the JavaOS for Business binary image at build time. These files are then available as part of the bootcode when JavaOS for Business boots. The bootcode is responsible for initiating communication with the server over standard network protocols, downloading

the image for the OS, identifying the characteristics of the system hardware to the boot image, and initiating the execution of the JavaOS for Business kernel. The mechanism that makes this possible is based on the Filizer tool, which is a custom tool that is part of the JavaOS for Business source release.

Adding files to the ROM file system requires adding entries to the HJLIBFILES list in **r1.0/client/build/javaos/java/java/GNUMakefile**. Files included in the ROM file system are organized into a hierarchical file system. By default, files are located in **/ROM**. For example,

```
HJLIBFILES= \
$(JAVAOSSRC)/lib/audio/ding.au \
```

includes an audio file in the ROM file system. Each entry in the HJLIBFILES list indicates the top-level directory where the file will be located in the ROM file system.

Prefix	Top-level directory
CWD	/ROM
JAVAOSSRC	/ROM
SHAREDSRC	/ROM
CLASSBINDIR	/ROM/lib

For example,

```
$(JAVAOSSRC)/lib/html/heapMap.html
```

represents

```
/ROM/lib/html/heapMap.html
```

This technique can be used to store small data files as well as large applications that have been ROMized with Jld in a JavaOS for Business binary image.

Using the Java linker (Jld)

Jld is a developer tool used by the JavaOS for Business build system to convert a series of class files into a single image that is bootable from ROM. Appendix A, “Tools” on page 29 describes Jld and how to use it.

Adding new source files

The JavaOS for Business source release is designed to build JavaOS for Business binaries for a Pentium-based PC. Licensees can use the sample implementation as a basis for new products. To build versions of JavaOS for Business to support new device drivers requires the addition of source files to appropriate directories and modification of the JavaOS for Business build script and makefiles.

The instructions described here are for modifying an existing build target (like PC) to include support for a new device driver. The task of modifying the build scripts to support a new platform is described in the next section.

The set of build files that are modified to support new features in JavaOS for Business are described in Table 14 on page 24. These are also described in Chapter 3, “Source code organization” on page 9.

Table 14. Build files	
File	Description
buildClient	Main JavaOS for Business build script. In most cases this script does not need modification.
Classfiles-<platform>.gmk	List of Java source files.
Defs-<platform>.gmk	Macro definitions.
Exportedfiles-<platform>.gmk	List of Java .class files used by javah to generate header files for native methods.
Rules-<platform>.gmk	Make rules.
GNUMakefile	C and assembly source files.

Here is how to add support for a new device driver for a new module named Snappy:

1. Change the current directory to the JavaOS for Business device driver directory.

```
cd /javaos/r1.0/client/src/javaos/sun/sun/javaos
```

2. Add the Java source code for the device driver to a file named Snappy.java.

JavaOS for Business device drivers are written in Java with native methods, if necessary. JavaOS for Business has a number of different kinds of device drivers. The *JavaOS for Business Device Driver Guide* describes how to write different kinds of JavaOS for Business device drivers.

3. [Optional] Change the current directory to the native method directory . If native methods are required they are placed in a separate directory.

```
cd /javaos/r1.0/client/src/javaos/sun/javaos
```

4. [Optional] Add the C or assembly source file for the device driver's native methods. Create a source file called Snappy.c.

5. Change the current directory to the JavaOS for Business build directory.

```
cd /javaos/r1.0/client/build/javaos/java/java
```

6. Modify the Classfiles-<platform>.gmk file to include the new Java source file Snappy.java.

This makefile is organized as a list of source files for the different components of JavaOS for Business. These include the common files as well as the platform-specific files (such as FILES-pc.java). In most cases, your new files will go into one of the platform-specific file lists.

7. Modify the Exportedfiles.gmk file to include the .class files for Snappy.

This file contains lists of .class files used by javah to generate C header files for native methods. Again, these lists are organized into groups. In most cases you will want to add your .class files to a product-specific group.

8. [Optional] Modify the file GNUMakefile to include the source files for the native methods.

This file contains lists of C and assembly source files for native methods. These are organized similarly to the Java source files in Classfiles.gmk. In most cases you will want to add your source file to a product-specific list like FILES-PC.o.

Adding a new platform

To add a new platform to the JavaOS for Business build system, you must create a build target directory and a platform-specific native method directory and modify the buildClient script and makefiles. The example below will create a new platform named Jelly. These instructions cover only the top-level procedures for adding a new platform to the JavaOS for Business build system. You should also study how the example directories are organized to gain a better understanding. The source directories are divided into three levels:

- *CPU-specific*. For example, the CPU-specific files for x86-based processors are listed in `/javaos/r1.0/client/build/javaos/java/java/Classfiles-i386.gmk`. For the example below, the CPU name is Grape.
- *Architecture-specific*. For example, the architecture-specific files for PC-based platforms are listed in `/javaos/r1.0/client/build/javaos/java/java/Classfiles-pc.gmk`. For the example below, the architecture name is Jelly.
- *Machine-specific*. For example, machine-specific files for PC-based platforms are listed in `/javaos/r1.0/client/build/javaos/java/java/Classfiles-PC.gmk`. For the example below, the machine name is KidStuff.

1. Create the CPU-specific directories for the sun.* package hierarchy.

```
mkdir -p src/javaos/sun.Grape/sun/javaos
```

2. Add the CPU-specific source files for the sun.* package hierarchy.

3. Create architecture-specific directories for the sun.* package hierarchy.

```
mkdir -p src/javaos/sun.Jelly/sun/javaos
```

4. Add the architecture-specific source files for the sun.* package hierarchy.

5. Create machine-specific directories for the sun.* package hierarchy.

```
mkdir -p src/javaos/sun.KidStuff/sun/javaos
```

6. Add the machine-specific source files for the sun.* package hierarchy.

7. Create the directory for holding the Java source files for platform-independent device drivers:

```
mkdir -p src/javaos/sun/sun/javaos
```

8. Add the Java source files for the platform-independent device drivers.

9. Create the directory for holding the C source files for the native methods for platform-independent device drivers:

```
mkdir -p src/javaos/sun/javaos
```

10. Add the C source files for the native methods for the platform-independent device drivers.

11. Modify the GNUmakefile to include the new native method source files.

12. Create the file for the list of new Java source files:

```
Classfiles-Grape.gmk  
Classfiles-Jelly.gmk  
Classfiles-KidStuff.gmk
```

13. Create the macro definitions files:

```
Defs-Grape.gmk  
Defs-Jelly.gmk  
Defs-KidStuff.gmk
```

14. Create the class list files:

```
Exportedfiles-Grape.gmk
Exportedfiles-Jelly.gmk
Exportedfiles-KidStuff.gmk
```

15. Create the make rules files:

```
Rules-Grape.gmk
Rules-Jelly.gmk
Rules-KidStuff.gmk
```

16. Modify **buildClient** to include the new build target.

Replacing the local or remote login authenticators

JavaOS for Business provides a local authenticator and a remote authenticator. A local authenticator authenticates a user to the network computer while a remote authenticator authenticates a user to the network itself. You can replace the local authenticator, remote authenticator, or both. You can use an authenticator that you write yourself or one you purchase.

The following topics describe how to replace the authenticators that JavaOS for Business provides with another authenticator. For information describing how to set up authentication, refer to *JavaOS for Business Network Operations*.

Replacing the local authenticator

To replace the local authenticator with your own authenticator:

1. Use the *unjar* tool to expand the login framework configuration archive file:
 - a. Get the LogonFw.jar file from the **/jars** subdirectory of the JavaOS Configuration Tool (JCT).
 - b. Extract the files from the JAR file in a clean, temporary directory using the following command:
2. Edit the properties file for the local authenticator that JavaOS for Business provides. The file name is: `com/ibm/joscfg/logonfwcfg/mri/SupportedLocalAuthenticators.properties`

```
jar -xf LogonFw.jar
```

3. The properties file is similar to the following:

```
supported.0=NSLAuthenticator
supported.1=HackAuthenticator
None.displayName=No default Authenticator
NSLAuthenticator.className=ibm.javaos.logonfw.NSLAuthenticator
NSLAuthenticator.displayName=NSL(Network Station Login)
```

```
HackAuthenticator.className=ibm.javaos.logonfw.HackAuthenticator
HackAuthenticator.displayName=Hack Login(For Dev & Test)
```

4. Add the information about your authenticator to the properties file. For example, suppose the file for your local authenticator has the fully qualified name `my.new.Authenticator.class`. In the JCT, your authenticator would be displayed as **My New Authenticator**.

You must add three lines to the properties file to identify your authenticator. Add the line `supported.2=NewAuthenticator` to the file, as shown below (in bold).

```
supported.0=NSLAuthenticator
supported.1=HackAuthenticator
supported.2=NewAuthenticator
```

Next, add two lines to the appropriate place in the file, as shown below (in bold):

```
HackAuthenticator.className=ibm.javaos.logonfw.HackAuthenticator
HackAuthenticator.displayName=Hack Login(For Dev & Test)
NewAuthenticator.className=my.new.HackAuthenticator
NewAuthenticator.displayName=My New Authenticator
```

5. Rearchive the JAR file:

- a. In the same temporary directory that you used to extract the files from the JAR file, create a file named `logonfw.mf`. Add the following lines to the file:

```
Name:com/ibm/joscfg/logonfwcfg/LogonFWConfig.class
Java-Bean: True
```

Note: The two lines you add to the `logonfw.mf` file will be the second and third lines of the existing `META-INF/MANIFEST.MF` file.

- b. Use the following command to *rejar* the tree. Leave out the existing `MANIFEST.MF` and provide the new manifest stub.

```
jar -cfm LogonFw.jar logonfw.mf com META-JCT
```

The `com` and `META-JCT` are the directories you created in step 1 on page 26. The **META-INF** directory was also created. However, do not *rejar* the **META-INF** directory because the `jar` utility creates the directory.

6. Replace the new `LogonFw.jar` file in the **JCT/jars** directory. When you select **Login Settings** using the JCT, you can now choose the new login authenticator.

Replacing the remote authenticator

To replace the remote authenticator with your own authenticator:

1. Use the *unjar* tool to expand the login framework configuration archive file:

- a. Get the `LogonFw.jar` file from the **/jars** subdirectory of the JCT.
- b. Extract the files from the JAR file in a clean, temporary directory. Use the command:

```
jar -xf LogonFw.jar
```

2. Edit the properties file for the local authenticator that JavaOS for Business provides. The file name is:

```
com/ibm/joscfg/logonfwcfg/mri/SupportedRemoteAuthenticators.properties
```

Note: The file name `SupportedRemoteAuthenticators.properties` is the only difference from replacing a local authenticator.

3. The properties file is similar to the following:

```
supported.0=NSLAuthenticator
supported.1=HackAuthenticator
None.displayName=No default Authenticator
```

```
NSLAuthenticator.className=ibm.javaos.logonfw.NSLAuthenticator
NSLAuthenticator.displayName=NSL(Network Station Login)
```

```
HackAuthenticator.className=ibm.javaos.logonfw.HackAuthenticator
HackAuthenticator.displayName=Hack Login(For Dev & Test)
```

4. Add the information about your authenticator to the properties file. For example, suppose the file for your local authenticator has the fully qualified name `my.new.Authenticator.class`. In the JCT, your authenticator would be displayed as **My New Authenticator**.

You must add three lines to the properties file to identify your authenticator. Add the line `supported.2=NewAuthenticator` to the file as shown below in bold.

```
supported.0=NSLAuthenticator
supported.1=HackAuthenticator
supported.2=NewAuthenticator
```

Next add two lines to the appropriate place in the file, as shown below in bold:

```
HackAuthenticator.className=ibm.javaos.logonfw.HackAuthenticator
HackAuthenticator.displayName=Hack Login(For Dev & Test)
NewAuthenticator.className=my.new.HackAuthenticator
NewAuthenticator.displayName=My New Authenticator
```

5. Rearchive the JAR file:
 - a. In the same temporary directory that you used to extract the files from the JAR file, create a file named `logonfw.mf`. Add the following lines to the file:

```
Name:com/ibm/joscfg/logonfwcfg/LogonFWConfig.class
Java-Bean: True
```

Note: The two lines you add to the `logonfw.mf` file will be the second and third lines of the existing `META-INF/MANIFEST.MF` file.
 - b. Use the following command to *rejar* the tree. Leave out the existing `MANIFEST.MF` and provide the new manifest stub.

```
jar -cfm LogonFw.jar logonfw.mf com META-JCT
```

The `com` and `META-JCT` are the directories you created in step 1 on page 27. The **META-INF** directory was also created. However, do not *rejar* the **META-INF** directory because the `jar` utility creates the directory.
6. Replace the new `LogonFw.jar` file in the **JCT/jars** directory. When you select **Login Settings** using the JCT you can now choose the new login authenticator.

Appendix A. Tools

The JavaOS for Business source release includes a few special tools to develop and test JavaOS for Business.

Jld - the Java class linker

Jld is a JavaOS for Business developer tool for linking Java class files. When a Java compiler compiles a Java source file, it generates a separate class file for each Java class in the source file. Such class files can then be loaded into a Java system; references to other class definitions can be resolved upon demand by the class loading and resolving mechanisms.

Managing multiple class files creates a certain amount of overhead. For applet developers, JDK 1.1 and later includes the jar archive utility, which combines several class files into a single jar file that can be transferred across a network with less overhead.

Jld performs a similar task for JavaOS for Business developers by providing two alternate means of class linking and constant pool resolution. The first method is to take a collection of class files and produce a single multiclass (mclass) file. The JavaOS for Business class loader can load this special class file format in a single operation. The mclass format can also be used as an intermediate format for the second method of class linking.

The second method is to generate an assembly code file that contains preloaded class data. The instructions in this assembly code file are not really machine instruction. They are mainly for laying out class data in a format for direct use by the Java virtual machine.

Mclass files can be used as input to another run of Jld or loaded directly into a running JavaOS for Business system. The assembly language files generated by Jld are target-system dependent and must be assembled and linked into the JavaOS for Business source code.

Note: While it is useful to compare Jld to jar, it should be noted that Jld is not a replacement for jar. In particular, the mclass file format is limited to JavaOS for Business development.

The two areas in JavaOS for Business development where Jld is used are the preparation of class files for inclusion in a JavaOS for Business image and with certain JavaOS for Business-specific applets or applications like HotJava Views. Mclass files can be used in external applications but the assembly code generated by Jld can be incorporated into JavaOS for Business only at build time.

Mclass files

Any set of arbitrary classes can be put into an mclass file, provided that for any class in the mclass, all of its superclasses—and the interfaces it implements are available. (A class is available if it resides in the same mclass file, is embedded in the ROM file system as a .class file, or is otherwise loadable through the CLASSPATH environment variable). The chief advantages of mclass files are:

- Reduced runtime memory footprint compared to class files (due to sharing of data between mclass classes)
- Smaller in size (less server space, faster loading)
- Module abstraction allowed for arbitrary collections of classes

Jld options

The **Jld** command has the following syntax:

```
java [java options] Jld [Jld options] filename ...
```

Note: **Jld** often needs an increased maximum heap size. For example, to increase the heap to 20 MB, type `java -mx20m`.

Option	Description
filename	Designates the name of a file to be used as input, the contents of which should be included in the output. Filenames are not modified by any pathname calculus. Filenames conventionally end with a .class or .mclass suffix, but this is not important to the operation of the program.
-o <i>outfile</i>	Designates the name of the output file to be produced. Conventionally, the file name ends with a suffix of .mclass for relocatable output (multiclass file format) or .s for assembly-language output, but this is not important to the operation of the program. In the absence of the -o option, an mclass file is produced with a name based on that of the first input file, stripped of pathname prefix and any suffix to which mclass is appended. This is often inappropriate.
-t	Enables generation of field and method tables. This causes the Jld to determine the offset of every instance variable and the method table offset of each nonstatic method for each class in the generated file. It requires that the complete inheritance hierarchy for each class be present in the set of linked classes. These tables are attached to each class as fieldtable and methodtable properties. This option should be used only on the ultimate link step.
-q	Enables transformation of method code to its quickened form. Many Java bytecode instructions refer to symbolic quantities such as the offset of a field or of a method, or simply to the name of a type. Normally, the Java virtual machine resolves these references upon execution and rewrites the instruction on the fly. This yields non-ROMable, and in a Solaris environment, non-sharable code. Java bytecodes resolved and quickened at link time are often read-only and thus both ROMable and sharable. Instructions referring to symbols that are not resolved remain unquick, and thus impure. This option should be used only on the ultimate link step. It implies -t.

Table 15 (Page 2 of 2). Jld options	
Option	Description
-qlossless	The same as -q, but leaves the resulting bytecode amenable to just-in-time compilation.
-c	Cumulative linking. Classes unresolved by the linking of class files explicitly listed as linker arguments are searched for the -classpath option and linked as they are found. File names are formed by concatenating the following: a path prefix, the character <code>java.io.File.separatorChar</code> (on UNIX, a <code>"/"</code>), the name of the class being sought, and the suffix <code>.class</code> .
-classpath <i>path</i>	Specifies the path Jld uses to look up classes. Directories are separated by <code>java.io.File.pathSeparatorChar</code> , which is typically a colon. Multiple classpath options are cumulative and are searched left to right. This option is used only in conjunction with the -c cumulative-linking option.
-v	Turns on the verbosity of the linking process. This option is cumulative. Currently, up to three levels of verbosity are understood by Jld. This option can be used as a debugging aid.
-r	Specifies that output is to be a (multiclass) mclass file. If -r is not specified, an assembly-language file is produced. The -r option is mutually exclusive with all of the following options: they are for use only when an assembly-language output file is to be produced.
-linenumbers	Enables writing of line-number tables in the output, if the information is available in the input data. These tables are not written by default. For use only when an assembly-language output file is to be produced.
-arch <i>target_architecture</i>	Designates the assembly language to be used in writing the output. The argument is <i>not</i> case sensitive. The only argument currently supported is SPARC. For use only when an assembly-language output file is to be produced. The SPARC output file writer also works for Intel x86 architectures when using the GNU assembler. Thus, any reference here to SPARC-specific or SPARC-only also applies to that system.
-imageAttribute <i>target-dependent_attribute</i>	Passes flags on to the target-specific output file writer. For use only when an assembly-language output file is to be produced. Table 16 on page 32 describes the attributes supported by the SPARC-specific output writer.

Retargeting Jld

Attribute	Description
<code>compiledCodeFlags=<i>number</i></code>	Specifies a numeric value (hex, octal, or decimal) to be placed in the <i>CompiledCodeFlags</i> field of each struct method block that is not mutable. See also <code>mutableCodeFlags</code> .
<code>mutableCodeFlags=<i>number</i></code>	Specifies a numeric value (hex, octal, or decimal) to be placed in the <i>CompiledCodeFlags</i> field of each struct method block that is mutable. See also: <code>compiledCodeFlags</code> , <code>mutableMBs</code> and <code>mutableMBClasses</code> .
<code>mutableMBs</code>	Locate all method-blocks in writable memory, rather than read-only memory where they normally reside.
<code>mutableMBClasses=<i>class list</i></code>	Locate method-blocks in writable memory for the classes specified. This option is cumulative and can be given multiple times to specify a list of classes. Class list is comma separated.
<code>JITInfoBufferSize=<i>number</i></code>	Specifies a numeric size (hex, octal, or decimal) in words of a buffer to be allocated for each Java method. The <i>CompiledCode</i> field of the struct method block points to this buffer. The buffer is word aligned. If this flag is not specified, or if the size specified is zero, then the buffer is not allocated and the <i>CompiledCode</i> field is set to zero.
<code>JITNativeInfoBufferSize=<i>number</i></code>	Specifies a numeric size (hex, octal, or decimal) in words of a buffer to be allocated for each native method. The <i>CompiledCode</i> field of the struct method block points to this buffer. The buffer is word aligned. If this flag is not specified, or if the size specified is zero, then the buffer is not allocated and the <i>CompiledCode</i> field is set to zero.

Jld is capable of producing target-specific assembly-language output that specifies JDK internal data structures to represent all the Java classes given Jld as input. In this section, the internal organization of the Jld is discussed and information is provided on retargeting its output to another system.

Internal organization

Jld is implemented in Java. The source code is partitioned into several packages located in the `r1.0/client/src/javaos/jld` directory. These classes are organized with the following conceptual scheme:

- Classes that model concepts of the Java language or class or mclass file formats, without reference to any particular Java runtime implementation. These classes are further divided into:
 - Classes common to both class and multiclass files found in package components
 - Classes representing the external layer of Jld, such as class and mclass files, and the Jld driver class, found in the anonymous package. A couple of generally useful classes, such as `BufferedPrintStream`, are also in this class.

- Classes that model concepts and data structures of the JDK runtime, without reference to target-machine representation. These classes are found in the `vm` package. (The most notable exceptions are class `Str2ID`, which is in the anonymous class and parts of class `Const`.)
- Classes with knowledge of the exact target-machine-dependent runtime representation of data structures are in the `coreimage` package. They are dependent on not only the target hardware but on the structure layout used by the C compiler to compile the Java runtime and the syntax of an assembler compatible with that compiler.

Choosing a target system

When it is producing an `mclass` file, `Jld` is completely target independent: neither class nor `mclass` file formats reference a specific platform. However, when the user desires a ROM-izable image of JDK internal data structures representing a set of classes (for example: when the `-r` command-line option is *not* chosen), it is necessary to choose a target system. This is done using the command-line option `-arch target_architecture`.

The default target architecture is SPARC but the process works for the PC as well, if GNU tools are used.

`Jld` does *not* have a table of target systems compiled into it. Instead, it uses Java's dynamic capabilities to construct the name of a class from a string, construct an instance of that class, and finally, ask it to write the output. The following code from `Jld.java` illustrates this process:

1. The default target architecture is SPARC:

```
String archName = "SPARC";
```

2. When `-arch` is seen, `archArg` is set to the next word in the command-line argument. Then, `archName` is set to an all-uppercase version of the same word:

```
archArg = clist[ ++i ];
archName = archArg.toUpperCase();
```

3. After processing the input files, `Jld` instantiates an output writer. It constructs the name of the writer using the `archName` variable and some fixed strings. It then attempts to load the class of that name:

```
String writername = "coreimage."+archName+"Writer";
Class writerClass = null;
try {
    writerClass = Class.forName( writername );
} catch ( ClassNotFoundException ee ){
    System.err.println("-arch "+archArg+" not supported");
    return false;
}
```

4. `Jld` instantiates an instance of that class. The class must implement the interface `coreimage.CoreImageWriter`.

```
CoreImageWriter w;
try {
    w = (CoreImageWriter)(writerClass.newInstance());
} catch ( Exception e ){
    System.err.println("Could not instantiate "+writername );
    e.printStackTrace( );
    return false;
}
```

5. The `init` method is called to finish the writer's initialization, passing two arguments that could not be part of the constructor's. The `setAttribute` method is called with each argument string seen by the driver following an `-imageAttribute` command-line option. This method should return true for recognized attribute strings, or false for unrecognized or malformed strings.

```
w.init(useLineNumbers, verbosity>0 );
Enumeration attr = attributes.elements();
while ( attr.hasMoreElements() ){
String val = (String)attr.nextElement();
if ( ! w.setAttribute( val ) ){
System.err.println("Bad attribute value "+val );
}
}
```

6. Several more calls are made to the output writer. This should cause the desired (assembly-language) file to be written to the file system, for whatever further processing is to be done by the system build process.

```
if ( w.open( outName ) != true ) {
w.printError( System.out );
good = false;
} else {
w.writeClasses( t );
w.printSpaceStats( System.out );
w.close();
}
```

Writing an output writer

When porting JavaOS for Business to the hal9000 architecture, for example, a class called `coreimage.HAL9000Writer` needs to be created. The easiest way to create the class is to start with the source for the `coreimage.SPARCWriter` class. Copy the class and as many of its auxiliary classes as needed. Change the names and presumably the output writing statements as you go.

Here are some important points:

- `coreimage.HAL9000Writer` must implement `coreimage.CoreImageWriter`, which is reproduced here:

```
package coreimage;
public
interface CoreImageWriter {
void init( boolean uselinenumbers, boolean
verbose );
boolean setAttribute( String attributeValue );
boolean open( String filename );
boolean writeClasses( components.ConstantPool
consts );
void printSpaceStats( java.io.PrintStream log );
void close();
void printError( java.io.PrintStream o );
}
```

- `coreimage.HAL9000Writer` must provide a public constructor of no parameters, so it can be constructed using `Class.newInstance()` as shown above.
- `coreimage.HAL9000Writer` may be defined as a subclass of `vm.JDKVM` though this is not strictly necessary. Nonetheless, you will want to use at least some methods from that class.
- `coreimage.HAL9000Writer` is not explicitly passed the list of classes. Instead, it is expected to rummage through global data structures, finding interesting things and writing them out. A vector of data structures representing classes to write out can be found by calling: `vm.ClassClass.getClassVector()`.

There are a number of global variable names that must be defined because the JDK uses them to traverse the data structures being written. These are described in Table 17 on page 35. The names given here match those given in C name space—your system names may require additional mangling.

Table 17. JDK global variables	
Global variable	Description
numRomClasses	An int cell representing the number of classes output, including any array classes you instantiate.
initialClassTable	An array of pointers to handles for class data structures. numRomClasses are included in this array.
ROMStringTable	This is a struct StrIDhash used to intern all the string literals that appeared in the classes linked. Keys are ASCII UTF8 zero-byte-terminated C strings. Associated objects are (pointers to) the handles for Java string objects. Notice that StrIDhash structures are chained together. The last one on the chain must be writable, so it can be modified in order to intern the string literals, which appear in classes loaded at runtime or using <code>java.lang.String.intern()</code> .
ROMHashTable	This is a StrIDhash struct containing the ASCII UTF8 zero-byte-terminated C strings for the class names and signatures that appear in many constant pool references. As above, the last structure on the chain must be writable.
classnameHandle	The root of the representation of each class is a struct referenced from Java so each must have a handle (as implemented by the current JDK). Many of these handles are referred to by name from C code in the JDK. Rather than try to keep track of them individually, just make all the handles global names. <code>classname</code> is the name as produced by <code>vm.ClassClass.mangleNames()</code> .

Filizer and Stuffer

Filizer and Stuffer are JavaOS for Business developer tools used to prepare data and class files for inclusion in the ROM file system. These tools are meant to be used within the JavaOS for Business build system.

JavaDoc

The JavaDoc tool is located in the `r1.0/client/src/share/sun/sun/tools/javadoc` directory of the JavaOS for Business source tree. Use this version of the JavaDoc tool to generate documentation from your source code. Be sure to use the `-public` and `-skipnodoc` parameters to produce the correct level of documentation.

Index

Special Characters

/home/userid/scripts/r1.0 directory 15
/pubs directory 5

Numerics

128-bit encryption 4, 17
40-bit encryption 4, 17

A

adding a new platform 25
aixtools.zip 4, 5
architecture-specific code 10
assembly language files 29
authenticators
 local 26
 remote 26
 replacing 26

B

bldlevel 4
boot image 5
boot mechanism 5
bootcode 22
booting the image 22
BSAFE 22
BufferedPrintStream 32
build environment
 configuring 5
 split-build 3
 system requirements 3
build options 20
build procedures 15
build target, modifying 23
build tool variables 15
build-time configuration parameters
 aixGNU 16
 aixJDK 16
 aixMach 15
 aixTargets 16
 buildFlags 16
 buildLevel 18
 buildRoot 15
 buildSecurity 17
 buildStyle 16
 buildSubLvl 18
 buildType 15
 CMVCRRelease 18
 copyCnfFile 17
 country 17

build-time configuration parameters (*continued*)

 driverName 18
 getExports 17
 intelGNU 16
 language 17
 mountDir 15
 publishClient 17
 publishJIE 17
 solarisGNU 15
 solarisJDK 15
 solarisMach 15
 solarisTargets 16
 srcLocation 15
 stgPubDir 18
 stgPubMnt 18
 stgServerHost 18
 toolsDirAix 15
 toolsDirSol 15
 userid 15
 vendorLongName 17
 vendorShortName 17
buildClient 7
buildClient script 15
buildServer 7
buildServer script 21
buildServer.cnf file 15, 21

C

checksum utility (cksum) 13
CLASSPATH environment variable 29
client source tree 10
CPU-specific code 10
Cygnus Solutions 4, 7

D

device drivers 14, 24
downloadable system services 4

E

encryption types 17, 22
error log daemon 4
error log viewer 4
Exportedfiles.gmk file 24

F

filizer 13, 35
filizer tool 23
fonts 4

G

- garbage collection 14
- GNU development tools 7
- GNUMakefile 12
- green threads 14

H

- heap management 14
- HJLIBFILES list 23
- HotJava Browser 4

I

- IBM Adaptation Kit for JavaOS for Business
 - CD content 3
- IBM Network Station Manager (NSM) 4
- index.htm 5
- Info-ZIP Web site 3
- Interactive Code Analysis Tool (ICAT) 4
- InterORB Protocol Enabler for Java (JIE) 4

J

- jar 29
- Java class files, linking 29
- Java class linker (Jld) 29
- Java linker (jld) 13
- Java runtime 14
- Java virtual machine 9, 29, 30
- JavaDoc 35
- JavaOS Configuration Tool (JCT) 4
- JavaOS System Database (JSD) 4
- JDK class library 9
- JDK hosting layer 9, 14
- JIE classes 10, 18
- jld 13
- josbind.zip 4
- josbine.zip 4
- josdbgd.zip 4
- josdbge.zip 4
- jossrtd.zip 3
- jossrce.zip 3
- jossrv.zip 10, 17, 18
- jsdk.zip 3

K

- kits, development
 - IBM Adaptation Kit for JavaOS for Business 1, 3
 - JavaOS for Business Software Development Kit (JSDK) 1

L

- local authenticator, replacing 26
- log, build output 11

M

- machine-independent code 10
- machine-specific code 10
- makefile directory 11
- makefile options
 - CDEBUG 21
 - CONSOLE_ENABLED 21
 - IPJOS 21
 - JDEBUG 21
 - NETSHELL 21
 - NOERRLOG 21
 - NOTRACE 21
- Master Configuration File (MCF) 5
- microkernel 9
- microkernel directory 14
- microkernel layer 9
- multiclass (mclass) file 29

N

- native method directory 24
- native methods 10, 14
- network computer 1
- ntsrvin.zip 4

O

- OpenCard Framework (OCF) 4
- OpenCard Framework Web site 4
- output writer 34

P

- PATH environment variable 5, 7
- Preboot Execution Environment (PXE) 5
- PXE DHCP flash code 4
- pxesrc.zip 4

R

- read-only memory (ROM) 22
- readme.htm 5
- remote authenticator, replacing 27
- ROM file system
 - adding files 23
 - HJLIBFILES list 23
- runtime layer 9

S

- Secure Sockets Layer (SSL) 21
- server source tree 10
- server, building 21
- setup script 7
- soltools.zip 4, 6
- source code 3
- source code organization 9
- source code tree 6
- Str2ID 33
- stuffer 35

T

- tools 29
 - filizer 13, 35
 - Java class linker (Jld) 29
 - JavaDoc 35
 - stuffer 35
- troubleshooting tools 4

U

- unzip utility 3

Z

- zip utility 3

