Using PersonalJava™ Solution for OS-9

Version 3.1

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044
International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620
RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision H
March 2001

Table of Contents

Chapter 1: PersonalJava Solution for OS-9 Overview

This chapter describes PersonalJava technology within an OS-9 system. It includes the following topics:

RadiSys.
MICROWARE SOFTWARE

What is Java?

In the early 1990s, programmers at Sun Microsystems realized the need for providing small, platform independent, secure, and reliable code for smart consumer electronics and settop boxes. Out of this need, the Java programming language was created.

To create the Java language, the designers began from the ground up. They borrowed some features from the most common languages used today, including C, C++, SmallTalk, and Common Lisp. The designers then added features like garbage collection and multithreading and threw out features like multiple inheritance, operator overloading, and pointers. What they created is an interpreted, object-oriented programming language portable to most network-based platforms including Windows®, Macintosh®, Unix®, and OS-9®, as well as an increasing number of settop boxes and electronic devices.

Java's features were selected because they were originally designed for the settop box market, which requires security and the ability to run code from untrusted hosts. In addition, Java has become the language of the Internet because these same features are important when downloading an application off the Internet.

PersonalJava and EmbeddedJava Technology

In 1997, Sun Microsystems created two subsets of Java to allow it to run more efficiently on devices with memory restrictions. PersonalJava Technology was created to work primarily on settop boxes (STB), PDAs (Personal Digital Assistant), screen phones, mid-range mobile phones, and web TVs. On the other hand, EmbeddedJava Technology was created to work in an even more limited working environment such as in industrial controllers/instrumentation, printers, pagers, and low-end mobile phones.

This manual documents the Microware implementation of PersonalJava Technology.

Why Java for OS-9?

OS-9 provides an ideal platform for Java because it is built around a robust process model and provides memory management and interprocess communication.

Many customers in the STB, network computer (NC), and wireless markets are looking for a real-time operating system (RTOS) that supports Java. Because of this, semiconductor manufacturers view Java as a way to differentiate their product and compete directly with the PC market.

Sun has defined subsets of Java for the embedded and personal computer. These subsets provide Microware with the means to make OS-9 the best software platform for Java-enabled devices.

PersonalJava Solution for OS-9

Microware has performed a variety of enhancements to Java to produce a PersonalJava Solution for OS-9 systems.

Enhancements to Java

The following changes have been made to improve Java's integration with OS-9:

- Java classes can be executed from a ROM module.
- The use of optional graphics features is user selectable.
- The Java Virtual Machine (JVM) runs as a process on OS-9. Multiple JVMs can be run simultaneously.
- A Java-enabled device can run completely diskless.
- The memory usage footprint can be computed exactly; this ensures that there are no out-of-memory errors after deployment.
- MAUI Applications can execute PersonalJava Applets in a window.
- Java Applications can execute MAUI applications in a window.

Loading Classes from JAR Files

PersonalJava Solution for OS-9 supports the use of Java Archive (JAR) files as defined in the Java Development Kit 1.1 documentation. These JAR files are useful because, like Java classes, they can combine other resources, such as HTML files and images. This enables all of the resources needed by a Java applet or application to be combined into a single JAR file.

For More Information

Refer to the Sun web page at http://java.sun.com for more information about the Java Development Kit 1.1.

Threading and Processing

Java is a multi-threaded application environment. Multi-threaded applications have the ability to interleave instructions from multiple independent execution threads (minimal processes).

Multi-threading also allows applications to perform multiple activities simultaneously such as processing input or loading graphics. If you have used a standard web browser, you are already acquainted with multi-threading. When you access a web page, you will notice that you can begin to scroll the page and read the text well before all the graphics have loaded. This is an example of multi-threading.

The Java API (Application Program Interface) provides a Thread class that supports a collection of methods to start, run, or stop a thread, as well as check on the status of a thread.

When writing Java applications, be sure to implement your classes and methods so they are thread-safe. If you want your objects to be thread-safe, any methods that may change the values of an instance variable should be declared synchronized. This ensures only one method can change the state of an object at any time.

For More Information

Refer to the book ***Concurrent Programming in Java*** by Doug Lea for more information on this subject.

Implementing Threads

OS-9 implements the JVM as a process. Java threads exist inside the process as native OS-9 threads. Thus, Java threads are scheduled right along with other OS-9 threads and processes.

For More Information

Refer to the ***OS-9 Technical Manual*** for more information on process/thread scheduling.

Preempting a Thread

Java’s threads are preemptive. This means that if a lower priority thread performs an action that wakes up a higher priority thread, the higher priority thread will execute instead of the lower priority thread.

Communicating With OS-9 Processes

A Java thread communicates with OS-9 processes including other JVMs either through named pipes and sockets or through native methods.

For More Information

Refer to the ***OS-9 Technical Manual*** for more information on using named pipes. Refer to ***Using LAN Communications Pak*** for more information on using sockets. Refer to ***The Java Programming Language*** by Ken Arnold and James Gosling or to ***http://java.sun.com*** for more information about native methods.

Memory Management

The OS-9 implementation of the JVM allocates memory from two basic areas: the Java heap and system RAM. The Java heap is allocated from system RAM when the JVM is first started; the maximum heap is allocated at that time. The maximum size of the Java heap is set using the -mx option. While the JVM is running, addition memory is allocated from system RAM as needed for such things as native thread stacks, GUI objects, class information, etc. The -ss command line option is used to set the size of the native thread stack.

For More Information

Refer to the Java help option for more information about the Java options available. Java help is available from the OS-9 command prompt by typing `pjava -help`.

Storage that is no longer being used in the Java heap is reclaimed using a mechanism called garbage collection. The JVM collects garbage when there is insufficient heap space to allocate an object or, if asynchronous garbage collection is enabled, at periodic intervals. The garbage collection mechanism does not affect or interact directly with the operating system.

Security

PersonalJava Solution for OS-9 V3.1 contains the security classes from JDK 1.2. These provide for the fine-grained security model so that an OEM can allow or disallow operations on an operation-by-operation basis.

Even though security functionality has been added, there are some security issues to keep in mind when developing Java applications for OS-9 once security functionality has been added. The JVM must be run with super-user privileges. This allows Java applications unlimited access to disk drives and other sensitive resources. This should be considered when allowing non-trusted applications to run or before removing restrictions from applets.

Note

The security property file that Microware ships is different that the default Sun version. Microware's version allows all operations. All operations are allowed to provide backward compatibility with previous versions of PersonalJava Solution for OS-9. Change the file `\MWOS\SRC\PJAVA\LIB\security\java.policy` to enforce a different policy. Refer to Sun JDK 1.2 documentation for the format of this file.

Chapter 2: PersonalJava Solution for OS-9 Environment

This chapter provides an overview of the PersonalJava Solution for OS-9 Environment.

This chapter includes the following topics:

Host System Architecture

The source and example code and makefiles for Java on the Windows (DOS)-based host are located in the directories as shown in **Figure 2-1**. <proc> indicates your processor family. <portproc> indicates the specific family member your port supports. <proc> and <portproc> may be the same. <port> indicates the board or family of boards your port supports.

Figure 2-1 Source File Directories for PersonalJava Solution for OS-9 on the Host

```
MWOS\DOS\BIN
MWOS\DOS\jdk1.1.8
MWOS\DOS\jdk1.1.8\bin
MWOS\DOS\jdk1.1.8\demo
MWOS\DOS\jdk1.1.8\include
MWOS\DOS\jdk1.1.8\lib
MWOS\SRC\AFW\WINMGR
MWOS\SRC\ASSETS\FONTS\AGFA
MWOS\SRC\DEFS
MWOS\SRC\PJAVA\DOC
MWOS\SRC\PJAVA\EXAMPLES
MWOS\SRC\PJAVA\LIB
MWOS\OS9000\SRC\DEFS
MWOS\OS9000\SRC\DEFS\JAVA
MWOS\OS9000\SRC\IO\MODMAN\FM\DESC
MWOS\OS9000\<proc>\ASSETS\FONTS\AGFA
MWOS\OS9000\<proc>\CMDS
MWOS\OS9000\<proc>\CMDS\BOOTOBJS
MWOS\OS9000\<proc>\LIB
MWOS\OS9000\<proc>\LIB\SHARED
MWOS\OS9000\<portproc>\PORTS\<port>\BOOTS\INSTALL\INI
MWOS\OS9000\<portproc>\PORTS\<port>\BOOTS\INSTALL\PORTBOOT
MWOS\OS9000\<portproc>\PORTS\<port>\CMDS
MWOS\OS9000\<portproc>\PORTS\<port>\CMDS\BOOTOBJS\PJAVA
MWOS\OS9000\<portproc>\PORTS\<port>\LIB\SHARED
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\JCC
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\MODMAN
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\RUNTIME
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\TARGET
```

The directories contain the following files:

`MWOS\DOS\BIN`	contains module and file archive building tools related to Java
`MWOS\DOS\jdk1.1.8`	contains the Windows JDK (Java Development Kit) 1.1.8 from Sun with a Microware-specific version of some files
`MWOS\DOS\jdk1.1.8\bin`	contains Java binary `.exe` and `.dll` files for the Windows host

Note

The original `javah.exe` and `javah_g.exe` found in JDK 1.1.8 have been replaced by versions that generate OS-9 headers.

`MWOS\DOS\jdk1.1.8\demo`	contains the JDK 1.1 demos from Sun for the Windows host

Note

The demo applets are provided as-is by Sun and may contain dependencies on sound assets. Also included, for historical purposes, are examples from JDK v1.0.2.

`MWOS\DOS\jdk1.1.8\include`	contains the Java 1.1 header files for the Windows host
`MWOS\DOS\jdk1.1.8\lib`	contains original files from Sun for JDK 1.1.8 with a Microware enhanced `jcc.zip` file for the Windows host

MWOS\SRC\AFW\WINMGR	contains a text version of the window manager settings file
MWOS\SRC\ASSETS\FONTS\AGFA	contains the raw TrueType (.ttf) and MicroType (.fco) files shipped with Microware's PersonalJava Solution. These files are converted into loadable data modules by the os9make file in this directory.
MWOS\SRC\DEFS	contains code required for compiling Microware shared libraries on the Windows host
MWOS\SRC\PJAVA\DOC	contains various documentation files
MWOS\SRC\PJAVA\EXAMPLES	contains the Non-JNI (Java Native Interface) and JNI native methods examples as well as other examples from Microware
MWOS\SRC\PJAVA\LIB	contains the JVM (Java Virtual Machine) properties files and class library zip files for the OS-9 target classes.zip should be used with the optimized VM (pjava) and classes_g.zip should be used with the debug VM (pjava_g).
MWOS\OS9000\SRC\DEFS	contains the modman header file
MWOS\OS9000\SRC\DEFS\JAVA	contains OS-9 specific header files for Java
MWOS\OS9000\SRC\IO\MODMAN\FM\DESC	contains the modman editmod description file
MWOS\OS9000\<proc>\ASSETS	contains font assets for the OS-9 target

`MWOS\OS9000\<proc>\CMDS`
contains JVM, window manager, and other support binaries for the OS-9 target

`MWOS\OS9000\<proc>\CMDS\BOOTOBJS`
contains the modman file manager and device descriptor

`MWOS\OS9000\<proc>\LIB`
contains code required for compiling shared libraries on the Windows host

`MWOS\OS9000\<proc>\LIB\SHARED`
contains shared PersonalJava Solution objects for the OS-9 target

`MWOS\OS9000\<portproc>\PORTS\<port name>\BOOTS\INSTALL\INI`
contains `JavaDemo.ini`, the demonstration Configuration Wizard configuration file

`MWOS\OS9000\<portproc>\PORTS\<port name>\BOOTS\INSTALL\PORTBOOT`
contains `java.ml`, the module list used when Java support is enabled in the Configuration Wizard

`MWOS\OS9000\<portproc>\PORTS\<port>\CMDS`
contains the window manager stock image resources

`stock_8.res`—8-bit bitmap and cursor support (default)

`stock_9.res`—16-bit bitmap and cursor support

Note

The filenames for the stock modules are in the following format:

`stock_<coding method>{_swapped}.res`

where `<coding method>` is the decimal value for the MAUI coding method used for the graphics device (8 = 8-bit CLUT, 9 = RGB555, etc.)

See `maui_gfx.h` for these values. `maui_gfx.h` is located in the following directory: `MWOS\SRC\DEFS\MAUI`.

The suffix `_swapped` is appended to stock files that have drawmaps nybble or byte swapped with respect to the target processor. For example, on a big endian processor the file called `stock_9_swapped.res` would contain images in 16-bit RGB555 that are byte swapped on 16-bit boundaries to be placed into little endian graphics memory.

`MWOS\OS9000\<portproc>\PORTS\<port>\CMDS\BOOTOBJS\PJAVA`
: contains the various port-specific files related to Java support

`MWOS\OS9000\<portproc>\PORTS\<port>\LIB\SHARED`
: contains a pre-generated color cube module for the Java `libmawt.so` shared library

`MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA`
: contains the makefile to run the makefiles in the sub-directories

These makefiles produce the files necessary to install Java on both disk-based or ROM-based systems.

`MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\JCC`
: contains the makefile to build `libclasses.so`, a pre-loaded version of `classes.zip` and classes from `classes.lst`.

`MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\MODMAN`
: contains the makefile used to build the modman archive of the properties files listed in `pjava_home.ml` and `pjava_home_g.ml`.

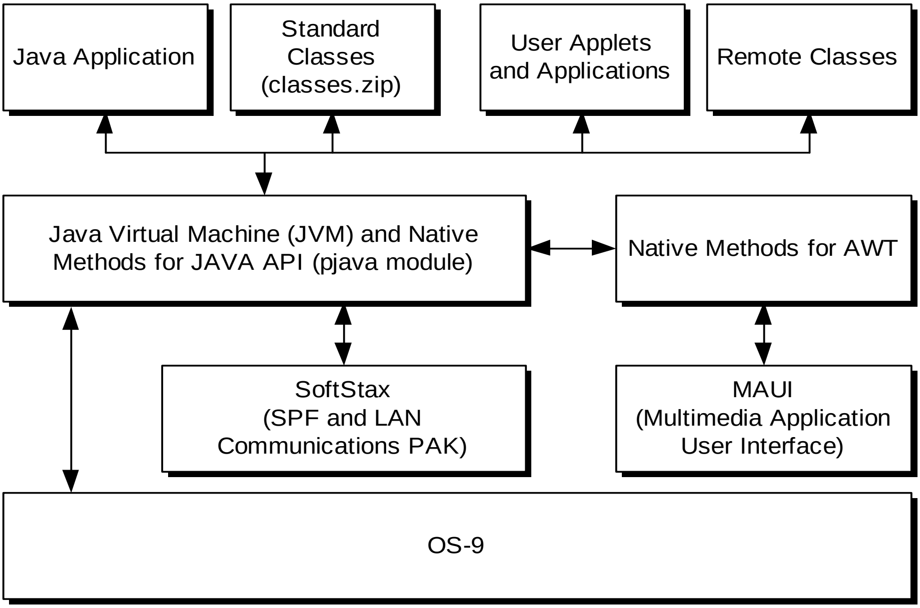
`MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\RUNTIME`
: contains the makefile used to build the merged module file suitable for adding to a boot, burning into Flash, or loading at run-time to get Java support

`MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\TARGET`
: contains the makefile used to build an archive suitable for transferring to a disk-based target and unarchiving to get Java support

PersonalJava Environment

Figure 2-2 shows how the modules and files representing the PersonalJava environment interact at runtime on an OS-9 based device.

Figure 2-2 The PersonalJava Environment



The Java Virtual Machine

The central component of the PersonalJava environment is the JVM contained in the `libjavai.so` shared library module. It reads Java class files that make up user applets and applications, third party applications, and classes that make up the standard Java Application Programming Interface (typically stored in the `classes.zip` file).

Native Methods and OS-9

Native Methods are methods for Java classes that are written in a native language, such as C or C++.

In the course of executing the methods of the Java classes, the JVM can be instructed to call Native Methods residing either within the JVM itself or in a user-defined shared library. These Native Methods can then make calls into one of the OS-9 File Managers (such as SoftStax), an OS-9 API support module (MAUI), or the OS-9 Kernel.

All native methods needed by the PersonalJava API (Application Program Interface) are contained in the `libjavai.so` and other shared library modules. These native methods make calls into SoftStax (for networking support), MAUI (for graphics, windowing, and audio), and the Random Block File Manager (RBF) and the Sequential Character File Manager (SCF) (for basic I/O). The core JVM makes calls directly into the OS-9 Kernel for such things as thread support and spawning processes.

For More Information

Refer to **Chapter 6:Creating Native Methods for OS-9** for more information about Native Methods.

I/O

PersonalJava Solution for OS-9 uses the standard input, output, and error paths of the process to implement the Java `System.in`, `System.out`, and `System.err` objects. The `System.in` object has special behavior, once I/O begins on that path the JVM is blocked until a carriage return character is encountered.

Support Files

`libmawt_0.dat` contains information about the color lookup table (CLUT) that is normally recalculated each time the Java graphics package is started. The recomputation is memory and CPU intensive and includes floating-point arithmetic. For systems that have only software floating-point, it can be very time consuming. For this reason, we have captured the output of the computations and included them in this module. The presence of this module in memory when `pjava` starts is optional, but it can decrease the start time and RAM usage dramatically.

This package includes a version of `libmawt_0.dat,` created for your package at the time it was developed. Due to circumstances outside the realm of Java, there may come a time when this module becomes out-of-date. There may be a time when Java starts up and it ignores `libmawt_0.dat` and creates `libmawt_1.dat`. This is not a bug; it simply indicates that your `libmawt_0.dat` module has become out-of-date.

If this occurs, follow these steps:

Step 1. Run Java and wait for it to create and finish the `libmawt_1.dat` module. When the module is complete it will have a correct CRC.

Step 2. Save the `libmawt_1.dat` module from memory using the save utility to a disk-based device and copy it to `MWOS/OS9000/<portproc>/PORTS/<portname>/LIB/SHARED`.

Step 3. Change the `loadjava` script located in `/h0/SYS` on your target to load `libmawt_1.dat` instead of `libmawt_0.dat`.

Note

Your .dat module only becomes out-of-date if other modules on the system are updated in such a way that the *color profile* of the system changes. In other words, if you do not update the MAUI or Java software in your system, the module may never become out-of-date.

Command Line Arguments

The following are the command line arguments and options for the `pjava` and `pjava_g` executables. <number> is a decimal number followed by ‘k’ to indicate kilobytes or ‘m’ to indicate megabytes or nothing to indicate bytes.

-help or -?	shows the summary of the command line options for your reference Print this message.
-bootclasspath <directories>	specifies the list of places to look for core classes. This specifies the set of directories, .zip, and/or .jar files in which to search for core class references. The directory names are colon delimited from each other. The boot class path defaults to `$JAVA_HOME\lib\classes.zip` or `classes_g.zip` when `pjava_g` is used.
-classpath <directories>	specifies the list of places to look for application classes This specifies the set of directories, .zip, and/or .jar files in which to search for application class references. The directory names are colon delimited from each other. See `-bootclasspath` for setting the location of the core classes (`classes[_g].zip`).
-D<name>=<value>	sets a system property This sets the name and value of a system property. These can be used instead of environment variables in many cases.
-debug	enables remote JAVA debugging

	The VM will print a password for the debug agent so a remote debugger can be attached.
-debugport<port>	specifies the debugger TCP/IP port number
	This specifies that a specific port be used for debugger communications. By default, a free port will be chosen automatically.
-fullversion	prints out the verbose build version
	This makes pjava print a verbose version string that reflects the implementation version and PersonalJava application environment version of Microware's PersonalJava Solution. Nothing more than printing the message happens.
-l<number>	sets the logging level
	This sets the verbosity of the logging messages printed by the VM. The higher the number, the more messages printed. The messages are printed using appdbg technology and can be viewed using `adump`. [Debug VM only]
-mr<number>	sets the red heap reserve size
	This sets the amount of memory remaining that indicates that the VM is critically low on Java heap memory. By default, heap reserves are disabled. Specify -mr and/or -my to enable heap reserves. See `sun.misc.VM` for using this value.
-ms<number>	sets the initial Java heap size
	This option is not supported in Microware's VM. -mx controls the size of the Java heap.
-mx<number>	sets the maximum Java heap size

	This is also the minimum Java heap size. The entire Java heap is allocated when the VM initializes so this value must be as large as the application will ever want it to be.
-my<number>	sets the yellow heap reserve size
	This sets the amount of memory remaining that indicates that the VM is running low on Java heap memory. By default, heap reserves are disabled. Specify -my and/or -mr to enable heap reserves. See `sun.misc.VM` for using this value.
-nm<number>	sets the number of extra monitors to expand monitor cache
	This sets the increment in number of cached monitors when the monitor cache underflows.
-noagent	suppresses use of libagent_g.so
	This option is not currently supported by Microware's VM.
-noasyncgc	disables asynchronous garbage collection
	The periodic automatic collection of garbage is suppressed. Use this option for the most fluid animations.
-noclassgc	disables class garbage collection
	Garbage collection of unusable class information is suppressed. This is for compatibility with VMs that don't garbage collect classes.
-noverify	does not verify any class
	This disables the act of class verification. This can pose a serious security risk if used.
-oss<number>	sets the maximum Java stack size for any thread

	This sets the Java stack size for threads within the VM.
-ss<number>	sets the maximum native stack size for any thread
	This sets the native (C) stack size for threads within the VM. See stackwatch for more information about determining native stack usage.
-t	turns on instruction tracing
	Information about each executed bytecode is printed. [Debug VM only]
-tm	turns on method tracing
	Information about each entered and exited method is printed. [Debug VM only]
-verbose or -v	turns on verbose mode
	Information on each class loaded and initialized is printed.
-version	prints out the build version
	This makes pjava print a terse version string that reflects the implementation version of Microware's PersonalJava Solution. Nothing more than printing the message happens.
-verbosegc	prints messages when garbage collection occurs
	Information about the amount of collected garbage and the amount of time it took to collect it is printed while the VM runs.
-verify	verifies all classes when read in
	This option enables the class verifier for all classes used by the VM.
-verifyremote	verifies classes read in over the network [default]

	This option enables the class verifier for only those classes that are loaded from remote network machines.
-Xrun<library>:<options>	execute a JVM extension module. Microware's PersonalJava currently has no supported JVM extension modules.

Environment Variables

The following environment variables are recognized by the JVM and its components:

`CLASSPATH`	specifies the location of the `.class, JAR` (Java Archive), or zip files used as application Java libraries. PJava loads core classes (those found in classes[_g].zip) from the directories specified with `-bootclasspath`. Adding a classes.zip to the `CLASSPATH` environment variable will have no effect. Use `-bootclasspath` or change the value of `JAVA_HOME` to change which classes.zip gets used. This can also be specified (and overridden) using the `-classpath` command line option. The locations are separated from one another with colons.
`HOME`	sets the `user.home` system property
`JAVA_HOME`	specifies the location of the Java properties files. This directory is also used as a basis for building the pathlist to the core class library .zip file. `classes.zip` should reside at `$JAVA_HOME\lib\classes.zip`. `classes_g.zip` should reside at `$JAVA_HOME\lib\classes_g.zip`. `pjava` uses `classes.zip`. `pjava_g` uses `classes_g.zip`.
`LD_LIBRARY_PATH`	specifies the locations of the shared library modules The locations are separated from one another with colons.

For diskless systems, the library modules can reside in memory and this variable need not be set.

`MEMWATCH`
: if set, the debugging version of the JVM emits memory use information when it terminates, or when `<cntl> C` is typed

The set value is not important. Refer to **Chapter 9:Monitoring PersonalJava Applications** for more information.

The set value is not important.

`MWOS`
: specifies the location of the MWOS directory on the platform

`PATH`
: specifies the locations to search for executable modules

The locations are separated from one another with colons.

`PORT`
: specifies the name of the terminal device

`SNDDEV`
: specifies the name of the sound device used by the JVM

`STACKWATCH`
: if set, the debugging version of the JVM emits stack usage information when it terminates or when `<cntl> C` is typed

The set value is not important.

For More Information

Refer to **Chapter 9:Monitoring PersonalJava Applications** for more information.

`TZ`
: sets the `user.timezone` system property

TZ is also used by Java’s native time and date functions.

For More Information

Refer to the ***Ultra C Library Reference*** manual for acceptable values for the TZ variable.

`USER` sets the `user.name` system property

Note

Refer to the ***Getting Started with PersonalJava Solution for OS-9*** manual for examples on how to set these variables for your target platform.

Chapter 3: Creating Java Applications for OS-9

This chapter provides an overview of PersonalJava Solution for OS-9 application development. It uses relatively simple exercises to describe development of Java applications and applets for an OS-9 target, with or without using AWT (Abstract Windowing Toolkit). More complicated Java application development, specifically Native Methods, is discussed in later chapters.

This chapter includes the following topics:

- **The Hello World Application (non-AWT Version)**
- **The Hello World Application (AWT Version)**
- **Tips for Running Your Application or Applet**

Note

During the installation process, the Sun JDK (Java Development Kit) 1.1.8 for Windows was installed in `\MWOS\DOS\jdk1.1.8` on your host. You must be using this version of the JDK while completing this tutorial. These examples use the E:\ directory. This location may vary depending on where you chose to install your PersonalJava Solution for OS-9 package).

For More Information

Make sure you have read the document ***Getting Started With PersonalJava Solution for OS-9*** and completed the exercises for the demos before beginning these exercises.

The Hello World Application (non-AWT Version)

Completing the exercise below will enable you to do the following:

- create a stand-alone Java application on your Windows 95/98/NT development host
- run the Java application on your Windows 95/98/NT development host
- run the same Java application on your OS-9 target system

Creating a Java Source File

The following example uses the non-AWT version of the `Hello World` application. The source is found in `E:\MWOS\SRC\PJAVA\EXAMPLES\HELLO\HelloWorldApp.java` on your host machine.

```
/**
* The HelloWorldApp class implements an application that simply displays "Hello
* World!" to the standard output.
*/
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Compiling the Source File

In a DOS shell on a Windows 95/98/NT development machine, compile the source file using the Java compiler.

```
> cd \MWOS\SRC\PJAVA\EXAMPLES\HELLO
> javac HelloWorldApp.java
```

When the compilation finishes, you will have a file named `HelloWorldApp.class` in the same directory as the Java source file.

Note

If the compilation failed, make sure you typed in and named the program exactly as shown in the example; it is case sensitive.

Running the Application on the Windows 95/98/NT Development Host

In a DOS shell on a Windows 95/98/NT development machine, run the program using the Java interpreter.

```
java HelloWorldApp
```

Hello World! is displayed to the standard output on the host machine.

Transferring the Class to the Target OS-9 System

In the below example, transferring the class to the target is done by using FTP. The transfer could also be done using NFS. To transfer your class to the target OS-9 system using FTP, complete the following steps:

Step 1. Choose `Start->Run` on the Windows desktop.

Step 2. In the Run dialog box, type `ftp <target machine name>` then click the OK button.

Step 3. Log on to the OS-9 machine by typing the user name and password in the FTP (MS-DOS Shell) window. The default user name and password for OS-9 machines are `super` and `user`.

Step 4. Change to the directory containing the application classes on the Windows machine by typing the following in the FTP window:

```
lcd MWOS\SRC\PJAVA\EXAMPLES\HELLO
```

Step 5. Change to the Java sources directory on the OS-9 machine by typing the following in the FTP window. Create the necessary directories if they do not exist. Your root disk device (/h0 in the example) may vary:

```
cd /h0/MWOS/SRC/PJAVA/EXAMPLES/HELLO
```

Step 6. Change to binary transfer by typing the following in the FTP window:

```
bin
```

Step 7. Transfer the class file by typing the following in the FTP window:

```
put HelloWorldApp.class
```

Step 8. Quit the FTP session once the transfer is complete:

```
quit
```

Starting the Java Application On the OS-9 System

To start the Java Application on your target using telnet to communicate with the OS-9 system, complete the following steps:

Step 1. Choose `Start->Run` from the Windows desktop.

Step 2. In the Run window text field enter `telnet <target machine name>` and click the OK button.

Step 3. Log onto the OS-9 system by typing the user name and password. `super` and `user` are the defaults for OS-9 systems.

Step 4. Change to the demo directory on the OS-9 machine by typing

```
chd /h0/MWOS/SRC/PJAVA/EXAMPLES/HELLO
```

Step 5. Run the application on the OS-9 target system using the Java interpreter:

```
pjava HelloWorldApp
```

Hello World! is displayed to the standard output on the target machine.

The Hello World Application (AWT Version)

The following example uses the AWT version of the `Hello World` application. The source is found in `MWOS\SRC\PJAVA\EXAMPLES\HELLO_AWT` on your host machine.

```
/**
 * The HelloWorldAppAWT class implements an application that displays "Hello
 * World!" using the AWT
 */
import java.awt.*;
import java.awt.event.*;

public class HelloWorldAppAWT extends Frame {
  Label statusBar = new Label();
  String status = "Hello World! ";
  WindowEventHandler weh = new WindowEventHandler();

  HelloWorldAppAWT() {
        super("Hello (AWT) Example");
        add("North", statusBar);
           addWindowListener(weh);
        setSize(300, 200);
           statusBar.setText(status);
        show();
  }

  public class WindowEventHandler extends WindowAdapter {

    public void windowClosing(WindowEvent evt)
    {
      System.exit(0);
    }
  }

    static public void main(String[] args) {
        new HelloWorldAppAWT();
  }
}
```

Follow the same steps as you did in **The Hello World Application (non-AWT Version)** on page 41 regarding compiling the source file; run the application on the windows host and transfer and run the application on your OS-9 target system with the following exceptions:

- replace the source file name `HelloWorldApp.java` with `HelloWorldAppAWT.java`
- replace the directory name `HELLO` with `HELLO_AWT`

- transfer the `HelloWorlAppAWT.class` and `HelloWorldAppAWT$WindowEventHandler.class` files to the target using FTP.

When the application is executed, a window appears containing the text **Hello World!** and a button to quit the Java application.

Tips for Running Your Application or Applet

Keep the following tips in mind after you have written your Java application or applet. Completing the steps below will help to assure that the applet or application will run on your target:

- Edit any class files that reference image (GIF and JPG) files, so that they reference the correct directory when loaded on your target. Below is an example:

```
imageViewer1.setURL(new java.net.URL("file:/h0/DEMO/img0001.gif"));
```

- Using FTP, transfer all class files, image files, and any other files needed to properly execute the application to the application's directory on the target. If you are running an applet, transfer any HTML files necessary to support the applet as well.

  If you are using a Java development tool such as Visual Cafe®, you may need to include additional `.zip` files that include class definitions for use with their tools. For example, Symantec's VisualCafe® has an additional `.zip` file called `Symclass.zip`. This can be found in the same directory as the `classes.zip` file.

Note

You cannot use Hawk to transfer these files because class files are not OS-9 modules.

- Update the `CLASSPATH` environment variable on the target to include the directory of your new application or applet or additional `.zip` files. This tells the JVM where to find classes it needs to run. Below is an example:

```
setenv CLASSPATH /h0/DEMO:$CLASSPATH
```

Chapter 4: Choosing a PersonalJava Diskless Strategy

This chapter continues the overview of PersonalJava application development for OS-9; it discusses two strategies for running your application on a diskless OS-9 target. The chapter also provides a practical example for each of the two strategies.

The following sections are included in this chapter:

RadiSys.

MICROWARE SOFTWARE

Introduction

During the implementation and debugging stages, the development scenario is similar to the one described in **Chapter 3: Creating Java Applications for OS-9**. This scenario is shown below:

Step 1. Create a Java source file.

Step 2. Compile the Java source file on the Windows host machine.

Step 3. Run and debug the Java application on the Windows host machine.

Step 4. Transfer the class or classes to the target OS-9 system.

Step 5. Run the Java application on the target OS-9 system.

Step 6. Debug/optimize how the Java application runs on the target OS-9 system.

After completing the above steps, it is time to select a diskless strategy for your diskless OS-9 target. The strategies you can pick from include the following:

- **Strategy 1: Adding Your Java Application to libclasses.so**
- **Strategy 2: Making the zip Files Into Data Modules**

For More Information

Refer to **Chapter 5: Additional Considerations for Choosing a PersonalJava Diskless Strategy** for an in-depth discussion of the pros and cons of both diskless target implementation strategies.

Source Files

The source files used to create Java applications for a diskless OS-9 target are shown below. Specific files are discussed on the following pages.

Figure 4-1 Source Files Used to Create Java Applications on a Diskless OS-9 Target

```
MWOS\DOS\jdk1.1.8\lib
   jcc.zip
MWOS\SRC\PJAVA\LIB
   classes.zip
   *.properties
   security\*
MWOS\OS9000\<portproc>\PORTS\<portname>\CMDS\BOOTOBJS\PJAVA
   libclasses.so
   pjava_home.mar
   pjava_home_g.mar
   pjruntime
   pjruntime_g
MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\JCC
   classes.lst
   makefile
MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\MODMAN
   pjava_home.ml
   pjava_home_g.ml
MWOS\OS9000\<portproc>\PORTS\<portname>\BOOTS
```

Strategy 1: Adding Your Java Application to libclasses.so

Listed below is information about building the merged module file pjruntime. Once built, this module contains the modules necessary to run PersonalJava Solution for OS-9 on a diskless system.

The Diskless PersonalJava Makefiles

The files below are included on the Windows host machine, in the directory `MWOS\OS9000\<portproc>\PORTS\<portname>\ PJAVA`. All relative pathlists shown below are relative to this directory.

`makefile`	calls the makefiles in the sub-directories
`JCC\makefile`	preloads the classes listed in `JCC\classes.lst` and the classes in `E:\MWOS\SRC\PJAVA\LIB\classes.zip` After pre-loading the classes, makefile calls the assembler and linker to produce the shared class library: `..\CMDS\BOOTOBJS\PJAVA\libclasses.so` Make changes in the JCC (JavaCodeCompact) directory if you want to change Java options, JCC options, or compiler options. If you choose to use pre-loaded classes, be sure to remove `classes[_g].zip` from `MODMAN\pjava_home[_g].ml`. This will avoid having two copies of the classes in memory.
`JCC\classes.lst`	contains no class files, as shipped

Edit this list if you want to add your Java application classes, zip archives, or jar archives to `libclasses.so`. For example:

```
..\..\..\..\..\..\SRC\PJAVA\MY_APP\project1\demo1.class
..\..\..\..\..\..\SRC\PJAVA\MY_APP\project1\demo2.class
..\..\..\..\..\..\SRC\PJAVA\MY_APP\project1\demo3.class
```

`JCC\jccargs` — shows a list of arguments to JCC

You should not have to edit this file.

For More Information

Refer to the document ***Using JavaCodeCompact for OS-9*** for an in-depth discussion of building a shared class library using JavaCodeCompact.

`MODMAN\makefile` — generates the modman archive of the PersonalJava Solution for OS-9 properties files and class zip archives listed in `MODMAN\pjava_home.ml` and `MODMAN\pjava_home_g.ml`

After running the mar utility, the makefile produces `..\CMDS\BOOTOBJS\PJAVA\pjava_home.mar` and `pjava_home_g.mar`.

`MODMAN\pjava_home.ml`
`MODMAN\pjava_home_g.ml` — list the files to convert into modman archives

Edit these if you want to add or change which files are put into the archives. Remove `classes.zip` if pre-loaded classes are being used.

For More Information

Refer to **Chapter 5: Additional Considerations for Choosing a PersonalJava Diskless Strategy** for an in-depth discussion of how to use the mar utility and the modman file manager.

RUNTIME\makefile	generates `pjruntime` and `pjruntime_g`, the merged module files containing complete PersonalJava support. This file can be very useful for diskless targets. After merging the modules the makefile produces `..\CMDS\BOOTOBJS\PJAVA\pjruntime` and `pjruntime_g`. Edit `pjruntime.ml` or `pjruntime_g.ml` to change to contents of these files.
TARGET\makefile	generates `pjava.mat`, the Microware Archive Tool archive containing all the files comprising PersonalJava support on a target machine. This file can be very useful for disk-based targets. After running mat (Microware Archive Tool) the makefile produces `pjava.mat` in the current directory. This file is then downloaded to a disk-based target and extracted with `mat` in the MWOS directory at the root of the system disk device. Refer to Appendix D: Microware Archive Tool for more information about using `mat`. Once installed, the target is capable of running PersonalJava applets or applications with modules and related files located on disk.

Running the Diskless PersonalJava Makefiles

Complete the following steps to run the diskless PersonalJava makefiles:

Step 1. Build a bootfile using the Configuration Wizard.

Step 2. Boot the target machine using this bootfile.

Step 3. Find the memory address of the kernel. At the OS-9 prompt, type the following:

```
$ mdir -e kernel
```

You should see something similar to the following:

```
            Current Module Directory

  Addr      Size       Owner     Perm Type Revs  Ed #  Lnk  Module name
-------- -------- ----------- ---- ---- ---- ----- ----- --------------
c002c520    83296      0.0      0555 Sys  a000    66     1 kernel
```

Take note of the memory address.

Step 4. Find the size of the bootfile.

Example: At the PC command prompt, type the following:

```
>cd E:\MWOS\OS9000\<portproc>\PORTS\<portname>\BOOTS\INSTALL\PORTBOOT
>dir os9kboot
```

You should see something similar to the following:

```
 Volume in drive E has no label.
 Volume Serial Number is 07CE-0507

 Directory of E:\MWOS\OS9000\ARMV4\PORTS\BRUTUS\BOOTS\INSTALL\PORTBOOT

08/12/98  09:54a             2,324,820 os9kboot
               1 File(s)      2,324,820 bytes
                            954,138,624 bytes free
```

Step 5. Convert the size of the bootfile to hexadecimal.

Example: 2324820 converts to 237954 in hex.

Step 6. Compute the memory address where `libclasses.so` will be placed. Use the following formula: kernel's address + size of bootfile = address for `libclasses.so`.

Example: c002c520 + 237954 = c0263e74

Step 7. Edit the line listed below in `E:\MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\JCC\makefile`.

```
#  Must be set to location where  libclasses.so  will be
#   in memory!
#
#   for this example, libclasses.so is the 1st module
#  loaded at address  0x30000000
#
MODULEBASE        =        0x30000000
```

MODULEBASE is the address at which `libclasses.so` resides. Change this value to the new address of `libclasses.so`.

Example: `MODULEBASE` = 0xc0263e80

For More Information

Refer to ***Using JavaCodeCompact for OS-9*** for an in-depth discussion of using the JavaCodeCompact.

Step 8. Set the Windows CLASSPATH environment variable so jcc.zip can be located and the PATH environment variable so the Windows JDK Java executables can be found. This is done on the Windows host machine by completing the following:

1. Right click on My Computer
2. Select Properties
3. Click on the Environment tab in the System Properties window
4. Select the CLASSPATH environment variable in either the System Variables or User Variable area
5. Enhance the value of CLASSPATH so it contains the path `%MWOS%\DOS\jdk1.1.8\lib\jcc.zip`
6. Click the `Set` button.
7. Click the `Apply` button.

8. Select the PATH environment variable in either the System Variables or User Variable area
9. Enhance the value of PATH so it contains the path `%MWOS%\DOS\jdk1.1.8\bin`
10. Click the `Set` button.
11. Click the `Apply` button.
12. Click the `Ok` button.

Your class path should look like the following:

```
%MWOS%\DOS\jdk1.1.8\lib\classes.zip;.;%MWOS%\DOS\jdk1.1.8\lib\jcc.zip
```

where %MWOS% is the MWOS directory in which you installed PersonalJava Solution for OS-9.

Note

If the application(s) you plan to run requires multiple windows, multiple frames, non-modal dialogs, menus or scroll bars, refer to **Chapter 8: Enhancing the Properties Files**, **Using Multiple Windows**.

Step 9. Run the PersonalJava makefiles. Type the following commands on the Windows host machine:

```
cd MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA
os9make
```

Note

In the pre-loading stage on the Windows host machine, the Java process loads all of the classes listed in `JCC\classes.lst` and `classes.zip` into memory. You may need to allocate more heap space for the Java process. To do this, edit the line listed below in `JCC\makefile`:

```
$(RDIR)/classes.a:      nulltrg ./$(MAKENAME)
   -$(DEL) $(RDIR)/classes.a
   java -mx48m JavaCodeCompact -f jccargs ...
```

Step 10. Paste a copy of libclasses.so into the directory containing os9kboot. Use the Windows Explorer to copy the following:

```
E:\MWOS\OS9000\<portproc>\PORTS\<portname>\CMDS\BOOTOBJS\PJAVA\libclasses.so
```

to

```
E:\MWOS\OS9000\<portproc>\PORTS\<portname>\BOOTS\INSTALL\PORTBOOT.
```

Step 11. Merge the bootfile file with `libclasses.so`.

Type the following command:

```
cd MWOS\OS9000\<portproc>\PORTS\<portname>\BOOTS\INSTALL\PORTBOOT

os9merge os9kboot libclasses.so > os9kboot_libclasses
```

Step 12. Transfer the bootfile merged with `libclasses.so` to the OS-9 target machine.

Rename the bootfile if necessary so it is the same name it had before `libclasses.so` was merged with it.

Example: `copy os9kboot_libclasses G:\os9kboot`

Note

If your target does not have a flashcard, add libclasses.so to your boot by adding it to `user.ml`. Refer to the Configuration Wizard help file for more information.

Step 13. Boot your target and type the following at the command prompt:

```
$ mdir -e libclasses.so
```

You should see something displayed similar to the following:

```
            Current Module Directory

  Addr      Size       Owner     Perm Type Revs  Ed #  Lnk  Module name
-------- -------- ----------- ---- ---- ---- ----- ----- --------------
c0263e80  1501600      1.0      0555 Subr 8001     7     2 libclasses.so
```

Step 14. Compare the memory address to the address you found in step 12. If they are the same, you have successfully loaded the romized classes.

Note

If any modules are added to the bootfile with the Configuration Wizard, you must repeat this process. Adding modules to the bootfile affects the address where libclasses.so is loaded.

Note

At the end of ***Getting Started with PersonalJava Solution for OS-9***, we introduced you to the `go.demo` script. You are welcome to add any or all of the following steps to this script.

For example, you can perform the following:

- enhance CLASSPATH for your application

- change the step to fork pjava so it runs your application instead of LaunchPad.

Example

The following example uses FTP to transfer pjruntime to the OS-9 target. Complete the following steps:

Step 1. FTP `pjruntime` from the PC to the OS-9 target.

Step 2. Load pjruntime by typing the following:

```
load -ld pjruntime
```

Step 3. Verify everything was loaded into memory by typing the following:

```
mdir
```

Step 4. Initialize the keyboard, mouse, and modman by typing (the device names for the keyboard or mouse may vary for your system)

```
iniz k0 m0 mm
```

Step 5. Set the JAVA_HOME environment variable by typing the following:

```
setenv JAVA_HOME /mm
```

Step 6. Start MAUI by typing the following:

```
maui_inp ^256 &
```

Step 7. Start the window manager by typing the following:

```
winmgr ^250 &
```

Step 8. Start your application by typing the following:

```
pjava <your application> &
```

Strategy 2: Making the zip Files Into Data Modules

If you choose not to build a shared class library using JavaCodeCompact you can use the properties files and the zip files containing the classes as data modules.

Listed below is information about building the mar archives `pjava_home.mar` and `pjava_home_g.mar` and making the zip files into data modules. After this process, you will have the modules needed to run PersonalJava technology on a diskless system.

The Diskless PersonalJava Makefiles

The files below can be found on the Windows host machine, in the directory `MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA`. All relative paths shown below are relative to this directory.

`MODMAN\makefile`	generates modman archives of the PersonalJava properties files and `classes.zip` listed in `MODMAN\pjava_home.ml` and `MODMAN\pjava_home_g.ml`. After running the mar utility, makefile produces `..\CMDS\BOOTOBJS\PJAVA\pjava_home.mar` and `pjava_home_g.mar`.
`MODMAN\pjava_home.ml` `MODMAN\pjava_home_g.ml`	list the files to convert into modman archives. Edit these files if you want to add or change which files are put into the archives.
RUNTIME\makefile	generates `pjruntime` and `pjruntime_g`, the merged module files containing complete PersonalJava support. This file can be very useful for diskless targets.

After merging the modules the makefile produces `..\CMDS\BOOTOBJS\PJAVA\pjruntime` and `pjruntime_g`.

Edit `pjruntime.ml` or `pjruntime_g.ml` to change to contents of these files.

For More Information

Refer to **Chapter 5: Additional Considerations for Choosing a PersonalJava Diskless Strategy** for an in-depth discussion of how to use the mar utility and the modman file manager.

Running the Diskless PersonalJava Makefiles

Note

If the application(s) you plan to run requires multiple windows, multiple frames, non-modal dialogs, menus, or scroll bars refer to **Chapter 8: Enhancing the Properties Files**, **Using Multiple Windows**.

Step 1. Run the PersonalJava makefile. Type the following commands on the Windows host machine:

```
cd MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\MODMAN
os9make
```

This makefile uses the Windows utility `mar` to make the file `pjava_home.mar` and `pjava_home_g.mar`, module archives of the PersonalJava properties files and classes.zip.

Step 2. Examine the resulting archive `pjava_home.mar` once the os9make finishes. Type the following on the Windows host machine:

```
ident -q ..\..\CMDS\BOOTOBJS\PJAVA\pjava_home.mar
```

The following modules are the properties files and classes.zip listed in `MODMAN\pjava_home.ml` that have been converted into a modman archive:

```
mm_tree     size #352      owner   0.0     ed #1    good crc #626274
mm_tree7    size #1718784  owner   0.0     ed #1    good crc #CD95B9
mm_tree6    size #2368     owner   0.0     ed #1    good crc #13263B
mm_tree5    size #8880     owner   0.0     ed #1    good crc #9B1B67
mm_tree4    size #5616     owner   0.0     ed #1    good crc #1F915B
mm_tree3    size #6032     owner   0.0     ed #1    good crc #194B0D
mm_tree2    size #1872     owner   0.0     ed #1    good crc #D2FE8E
mm_tree1    size #1712     owner   0.0     ed #1    good crc #CF9C7F
```

Creating the Data Modules For Your Application

Step 1. Go to the directory on the Windows development host containing your zip or JAR file. For this example, we will assume it's called java_app.zip

```
cd MWOS\SRC\PJAVA\JAVA_APP
```

Step 2. Run the mkdatamod utility by typing the following:

```
mkdatmod java_app.zip -to=os9000 -tp=<proc>
   java_app.zip.mod -n=java_app.zip
```

Step 3. Run the Microware ident utility by typing the following:

```
ident java_app.zip.mod
```

Step 4. Examine the module information for java_app.zip.mod. It looks similar to the following:

```
Header for:       java_app.zip
Module size:      $b0b0        #45232
Owner:            1.0
Module CRC:       $E10C02      Good CRC
Header parity:    $9175        Good parity
Edition:          $1           #1
Ty/La At/Rev      $400         $8000
Permission:       $111         -------r---r---r
Data Mod, Sharable
```

Note

The file name `java_app.zip.mod` was used in this example. Note that `java_app.zip.mod` contains the `java_app.zip` data module which is different than the original `java_app.zip` file that contains your application's zipped classes.

Step 5. Add the data module names that contain your zipped classes to `MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\RUNTIME\pjruntime.ml`. See the below example:

```
..\..\..\..\OS9000\SH3\ASSETS\FONTS\AGFA\MT\mwp_java.fco
..\..\..\..\OS9000\SH3\ASSETS\FONTS\AGFA\TT\utt.ss
CMDS\BOOTOBJS\PJAVA\pjava_home.mar
* java_app.zip.mod - data module containing the classes for my Java
* application
*
..\..\..\..\SRC\PJAVA\JAVA_APP\java_app.zip.mod
```

Step 6. Run the make file to merge the modules needed to run PersonalJava Solution on a diskless system into the file pjruntime.

Type the following commands on the Windows host machine:

```
cd MWOS\OS9000\<portproc>\PORTS\<portname>
    \PJAVA\RUNTIME
os9make
```

Step 7. After the make finishes, examine the resulting file `pjruntime`. Type the following on the Windows host machine:

```
ident -q ..\..\CMDS\BOOTOBJS\PJAVA\pjruntime
```

The following modules needed to run PersonalJava Solution on a diskless system are printed out:

```
stock_8.res size #18432   owner   1.0     ed #1    good crc #5FA254
winmgr     size #397648   owner   0.0     ed #13   good crc #469219
winmgr.dat size #2896     owner   0.0     ed #1    good crc #88030B
pjava      size #34128    owner   1.0     ed #30   good crc #9F84BF
libjavai.so size #651848  owner   1.0     ed #30   good crc #A4C20B
libjavafile.so size #23112 owner  1.0     ed #30   good crc #770146
libmawt.so size #1317832  owner   1.0     ed #30   good crc #323166
libmawt_0.dat size #33600 owner   0.0     ed #1    good crc #5AB16B
libnet.so  size #55064    owner   1.0     ed #30   good crc #E4E0F6
libzip.so  size #48152    owner   1.0     ed #30   good crc #FE64B1
mm         size #200      owner   0.0     ed #1    good crc #86EFE3
modman     size #9032     owner   1.0     ed #12   good crc #6EC6CF
umt.ss     size #608      owner   0.0     ed #1    good crc #78E4C6
mw_java.fco size #117136  owner   0.0     ed #1    good crc #C6E3D4
mwp_java.fco size #4512   owner   0.0     ed #1    good crc #671DDE
utt.ss     size #608      owner   0.0     ed #1    good crc #5E46CD
mm_tree    size #352      owner   0.0     ed #1    good crc #626274
mm_tree7   size #1718784  owner   0.0     ed #1    good crc #CD95B9
mm_tree6   size #2368     owner   0.0     ed #1    good crc #13263B
mm_tree5   size #8880     owner   0.0     ed #1    good crc #9B1B67
mm_tree4   size #5616     owner   0.0     ed #1    good crc #1F915B
mm_tree3   size #6032     owner   0.0     ed #1    good crc #194B0D
mm_tree2   size #1872     owner   0.0     ed #1    good crc #D2FE8E
mm_tree1   size #1712     owner   0.0     ed #1    good crc #CF9C7F
java_app.zip size #45232  owner   0.0     ed #1    good crc #E10C02
```

Step 8. Put the `pjruntime` file on your diskless OS-9 target by whatever means are appropriate for your target. For example, the `pjruntime` can be put into a bootfile, transferred using FTP to a RAM disk, loaded from the network using NFS, or burned into the FLASH EPROM by third-party tools.

Note

At the end of ***Getting Started with PersonalJava Solution for OS-9***, we introduced you to the `go.demo` script. You are welcome to add any or all of the following steps to this script.

For example, you can perform the following:

- enhance CLASSPATH for your application
- change the step to fork `pjava` so it runs your application instead of `Launchpad`.

Example

The following example uses FTP to transfer pjruntime to the OS-9 target. Complete the following steps:

Step 1. FTP pjruntime from the PC to the OS-9 target.

Step 2. Load pjruntime by typing the following:

```
load -ld pjruntime
```

Step 3. Verify everything was loaded into memory by typing the following:

```
mdir
```

Step 4. Initialize the keyboard, mouse, and modman by typing the following:

```
iniz k0 m0 mm
```

Step 5. Set the JAVA_HOME environment variable by typing the following:

```
setenv JAVA_HOME /mm
```

Step 6. Set the CLASSPATH environment variable by typing the following:

```
setenv CLASSPATH /mm/java_app.zip:.
```

Step 7. Start MAUI (Multimedia Application User Interface) by typing the following:

```
maui_inp ^256 &
```

Step 8. Start the window manager by typing the following:

```
winmgr ^250 &
```

Step 9. Start your application by typing the following:

```
pjava <your application> &
```

Chapter 5: Additional Considerations for Choosing a PersonalJava Diskless Strategy

The Java Development Kit (JDK), as shipped from Sun, is targeted strictly at desktop environments. Even the new PersonalJava technology aimed at consumer devices like cell phones and PDAs (Personal Digital Assistant), requires a file system for items such as the properties files and copyright information. Because OS-9 developers are typically building devices with no file system, Microware has developed a mechanism for simulating a file system in ROM. The mechanism uses ROM to implement the file system since RAM memory is usually limited on consumer devices.

This chapter discusses the requirements for running Microware's PersonalJava Solution on a diskless target, the items to consider when choosing a diskless strategy, and the use of the modman file manager. It also provides an example of using PersonalJava technology on a diskless target.

The following sections are included with this chapter:

- **Diskless Target Requirements**
- **Diskless Target Implementation Strategy**
- **Using the modman File Manager**
- **Diskless Target Example**

Diskless Target Requirements

Below is a list of requirements for using Java on a diskless target.

Java Requirements

This section describes the files Java expects to find in a file system at runtime. These files contain definitions of Java properties which are similar in use to environment variables.

`LIB/appletviewer.properties`
: contains definitions of various strings and settings used by the AppletViewer

`LIB/awt.properties`
: specifies various keys used to provide AWT (Abstract Windowing Toolkit) functionality

`LIB/content_type.properties`
: specifies various file content types, extensions for files containing such content and any applications that deal with those files

`LIB/font.properties`
: specifies which platform specific font resources to use to implement a particular Java font

`LIB/remote_classes.properties`
: specifies parameters for loading classes from a remote server

`LIB/security/java.security`
: contains settings used by the `java.security` package

`LIB/security/java.policy`
: contains the enforced security policy. Refer to JDK1.2 documentation for the format of this file.

`LIB/classes.zip` is a zip archive of the Java class files making up the PersonalJava API (Application Programming Interface)

This file can be customized to contain a subset of the complete PersonalJava API and can also be customized to contain OEM application code.

Note

The `classes.zip` file is not required if a rommed class module containing all needed classes is present. Refer to ***Using JavaCodeCompact for OS-9*** for more information about rommed class modules. The location of the `classes.zip` file is specified by the CLASSPATH environment variable.

For More Information

Refer to Chapter 10: Working with Remote Classes for more information on using remote classes.

All of the previous items should be located in the subdirectory `LIB` relative to the directory specified by the environment variable `JAVA_HOME` or the `java.home` property.

For More Information

Refer to **Properties vs. Environment Variables** on page 75 of this chapter for more information about this subject.

Window Manager Requirements

The Microware window managers require the data module `winmgr.dat` that contains settings for the window managers. This data module should be included in your bootfile.

For More Information

Refer to Chapter 7: Using the Window Manager for more information about the window managers.

Diskless Target Implementation Strategy

In order to meet the requirements of Java and the window manager, it is necessary to provide a disk-like interface to objects located in ROM. The modman file manager provides such an interface. Using the module archive utility (`mar`), a directory hierarchy, existing on a development system, can be converted to a number of OS-9 modules. These modules can be added to the target system's bootfile where they can be treated as members of a normal file system by way of the modman file manager. Details of using modman and the `mar` utility are given below along with example packaging scenarios.

Class Storage Options

When using PersonalJava Solution on a diskless system, there are several ways you can provide the Java classes that comprise the Java API. The three most attractive alternatives include 1) converting the Java class files into an OS-9 module by a process called prelinking that can then be loaded into ROM or RAM, 2) including a `classes.zip` file in the modman archive, or 3) using remote class loading.

For systems where RAM power requirements and costs are prohibitive, the rommed class approach makes the most sense. Since romized classes can be used directly from ROM, no RAM is needed to store the class information. Prelinking the entire PersonalJava class set results in a RAM savings of approximately one megabyte.

When ROM memory space is limited and RAM space is relatively plentiful, or in situations where ROM access times are slow, it is suggested to use a `classes.zip` file stored in the modman archive. A compressed `classes.zip` file is approximately one-half the size of a corresponding set of romized classes. Classes loaded from a `classes.zip` file are placed into RAM memory. This allows the JVM (Java Virtual Machine) to access the class data faster than it could if the class data were in ROM.

For applications with access to a remote server, classes can be loaded over the network. Having classes on a server system simplifies application updates because only the classes on the server need to be updated.

However, the basic Java packages `java.lang`, `java.io` and `java.util` cannot be loaded remotely. They must reside on the system either as rommed classes, a `.zip` file, or from unzipped class files.

Note

Classes loaded from a remote server require as much RAM as those loaded from a `.zip` file.

For More Information

Refer to ***Using JavaCodeCompact*** for more information about rommed class modules. Refer to ***Chapter 10: Working with Remote Classes*** for more information on working with remote classes.

Properties vs. Environment Variables

On a disk-based system, several parameters to the JVM are passed by way of environment variables. These include the `CLASSPATH` and `JAVA_HOME` environment variables. On a diskless system, however, environment variables can be inconvenient since they typically need to be set by a shell. Java properties are a useful alternative to environment variables. Java properties can be specified on the command line that invokes the JVM; this is done using the syntax -D<var-name>=<value>.

The classpath is a special case. The command line option, `-classpath`, is used to specify the classpath. As an example, consider the following two sets of commands:

```
setenv CLASSPATH /mm/LIB/java_app.zip
setenv JAVA_HOME /mm
pjava com.company.MyApp
```

The above is equivalent to the command line below:

```
pjava -Djava.home=/mm -classpath /mm/LIB/java_app.zip
com.company.MyApp
```

The latter form is useful since it can be used as the initial process for the system to run. The form using environment variables could only be used in conjunction with a shell.

Using the modman File Manager

The following section provides instructions for using the modman file manager.

Generating the modman Archive

In order to successfully execute Java in a diskless environment, a series of modules must be put into the bootfile for the target system. This series of modules is generated into a single loadable file called, `pjava_home.mar` by the utility `mar` (module archive) on the PC. The command line that generates the archive should be executed in the following directory:

E:\MWOS\SRC\PJAVA:

```
mar -tp=<proc> -o=E:\MWOS\OS9000\<port_proc>\PORTS\
    <port_name>\CMDS\BOOTOBJS\PJAVA\pjava_home.mar
    -z=E:\MWOS\OS9000\<port_proc>\PORTS\<port_name>\
    PJAVA\MODMAN\pjava_home.ml
```

Adding the modman Archive to the Boot

The modman archive (`MWOS\OS9000\<proc>\PORTS\ <port_name>\CMDS\BOOTOBJS\PJAVA\pjava_home.mar`) as well as the modman file manager (modman) and device descriptor (`mm`) should be added to the bootfile for the diskless device in whatever manner other modules are added to the boot. For disk-based devices, they can be loaded after the system is started.

For More Information

Refer to your target system's ***Board Guide*** for more information about creating boot options for your target.

Initializing modman

We recommend that modman be initialized prior to executing Java applications. This can be done either by using the `iniz` utility or programmatically by using the `_os_attach()` system call. Although this initialization is not absolutely necessary, performance may suffer if it is not done.

Example: Type `iniz mm` at the OS-9 prompt.

Setting the JAVA_HOME Environment Variable

The `JAVA_HOME` environment variable should be set to `/mm` before executing the JVM on a diskless target. This can be accomplished by using the `setenv` shell command or programmatically by using `putenv()`.

Using the mar Utility

Part of building the bootstrap for an embedded system requires using a tool that converts directories along with all their subdirectories and files into a single output file. This output file can be included in an OS-9 bootstrap file. The file also contains modules for each file in the directory and its subdirectories, as well as an additional module that contains information on how to create a directory structure to match the original in the module directory. This last module is linked by modman's file manager initialization code. The name of the module is specified in the device descriptor. Any number of these conversion modules may be specified.

For More Information

For more information on the `mar` utility, refer to the ***Utilities Reference*** Manual.

Diskless Target Example

This example configures PersonalJava Solution to run on a StrongARM machine without a disk.

Building the modman Archive

For this example, you need to build a modman archive containing the properties files required by PersonalJava Solution for OS-9. The package supplies the properties files in the directory `MWOS\SRC\PJAVA\LIB`. Because the PersonalJava runtime, `pjava`, expects to find the properties files in the directory `LIB` relative to the Java home directory, it is important to include the `LIB` directory as part of the archive. The property files to be included in the archive are listed in a file called `pjava_home.ml`. The contents of this file are listed below:

```
* Edit this file as appropriate to include/exclude
* files in/from ROMable image
LIB/appletviewer.properties
LIB/awt.properties
LIB/content_types.properties
LIB/font.properties
LIB/remote_classes.properties
LIB/security/java.security
LIB/security/java.policy
LIB/classes.zip
*LIB/javamath.zip
*LIB/javasql.zip
*LIB/javarmi.zip
*LIB/sunrmi.zip
*LIB/JCCMessage.properties
*LIB/JDCMessage.properties
```

The file `pjava_home.ml` is located in the directory E:\MWOS\OS9000\<proc>\PORTS\<port_name>\PJAVA\MODMAN on your Windows host.

The following command line command generates the following archive:

```
MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA> mar -tp=<proc> -o=..\CMDS\
    BOOTOBS\PJAVA\pjava_home.mar -z=MODMAN\pjava_home.ml
```

Chapter 6: Creating Native Methods for OS-9

This chapter provides an overview of native methods. It includes the following topics:

Note

During the installation process, Sun's JDK (Java Development Kit) 1.1.8 for Windows was installed in \MWOS\DOS\JDK1.1.8 on your host. You *must* be using this version of the JDK while completing this tutorial. These examples use the `E:\` directory. The location may vary depending on where you chose to install your PersonalJava Solution for OS-9 package).

MICROWARE SOFTWARE

For More Information

Be sure to read the document ***Getting Started With PersonalJava Solution for OS-9*** and complete the exercises for the demos before beginning this tutorial. In addition, you may wish to read **Chapter 3: Creating Java Applications for OS-9** and complete that tutorial before proceeding.

Using Native Methods on OS-9

Overview

This tutorial describes how to implement and debug native methods on the OS-9 operating system. The example given here implements a Java application to display and set the system time on an OS-9 machine.

Note

You will go through this tutorial twice. The first pass uses non-JNI (Java Native Interface) Native Methods. The second pass uses JNI Native Methods.

The subject of native methods is a very broad topic. This tutorial does not attempt to cover all the issues involved in implementing native methods but instead covers only those aspects of the process unique to OS-9.

For More Information

For a more complete discussion of the topic of Native Methods, please refer to ***The Java Programming Language***.

Requirements

You should have a basic understanding of the Microsoft Windows interface. You should already know how to navigate Explorer, how to select items using the mouse, and how to use drag and drop. In addition, you should also have a fundamental understanding of the Java programming language including the Abstract Windowing Toolkit (AWT) and native methods.

For More Information

Refer to Appendix E: Sources of Information for a list of references on these subjects.

Finally, you should be familiar with the Microware Hawk development environment.

For More Information

You must use Hawk for this example. Refer to ***Using Hawk***.

Objective

The objective of this tutorial is to show how to implement native methods on OS-9 and how to debug native methods using the Hawk development environment. To achieve this objective, complete the following:

- **Write the Application**
- **Add the Native Methods**
- **Run the Example Application on the OS-9 System**
- **Debug the Native Methods**

Write the Application

Write a Java application using native methods to perform certain functions.

Add the Native Methods

To add native methods, complete the following:

- add the declarations for the native methods
- generate the header file for the class containing the native methods
- generate the stub file used to create the native method shared library
- generate the export table source file used to create the native method shared library
- add the native method code
- compile and link the native method shared library
- add calls to the native methods into the Java class
- add a static initializer to load the shared library

Run the Example Application on the OS-9 System

To run the example application, complete the following:

- transfer the class files to the OS-9 system
- start the Java application

Debug the Native Methods

To debug the native method, complete the following:

- attach to the Java process
- attach to the shared library
- set breakpoints in the native methods

Using JNI Native Methods

Once you have completed the non-JNI native method tutorial, repeat the tutorial for JNI native methods:

- generate the JNI header file for the class containing the native methods
- generate the export table source file used to create the JNI native method shared library
- add the native method code
- compile and link the native method shared library

Environment

Windows 95, 98 or NT 4.0 is the host operating system. The source files for the example are in the Windows `MWOS\SRC\PJAVA\EXAMPLES\ NATIVE` directory on the distribution CD-ROM. If the PersonalJava Solution for OS-9 package has been installed, the source files are in the `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE` directory on a Windows disk device.

Writing the Application

The first step in implementing this example Java application is to write the Java classes used in this application. This can be done by using a text editor, but it can also be done by using a Java development environment. You will find four source files in your `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI` directory. They comprise the example program. Before moving on, read through the code listings and descriptions below.

The *TimeApp* class

The first class is the application called `TimeApp` and is located in `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI`. The application extends the Java `Frame` class. The source for the `TimeApp` class is shown below:

```
// TimeApp - example Java application using native methods to display and
//           set the system time on an OS-9/OS-9000 system.
//
package time;
import java.awt.*;
class TimeApp extends Frame
{
  // constructor for TimeApp class
  //
  public TimeApp()
  {
    super("Time Application");

    setLayout(null);
    resize(insets().left + insets().right + 220, insets().top + insets().bottom +
95);
    timeDisplay = new TextField(20);
    getTimeButton = new Button("Update");
    setTimeButton = new Button("Set...");
    quitButton = new Button("Quit");
    add(timeDisplay);
    timeDisplay.reshape(10, 30, 190, 24);
    add(getTimeButton);
    getTimeButton.reshape(10, 60, 50, 20);
    add(setTimeButton);
    setTimeButton.reshape(70, 60, 50, 20);
    add(quitButton);
    quitButton.reshape(130, 60, 50, 20);
```

```
    crntTime = new SysTime();
    timeDisplay.setText(crntTime.toString());
  }

  // event handler for TimeApp class
  //
  public boolean handleEvent(Event event)
  {
    // look for possible button events
    //
    if (event.id == Event.ACTION_EVENT) {
      if (event.target == getTimeButton) {
        clickedGetTime();
        return true;
      }
      else if (event.target == setTimeButton) {
        clickedSetTime();
        return true;
      }
      else if (event.target == quitButton)
        System.exit(0);
    }
    return false;
  }

  // handle updating the display
  //
  private void clickedGetTime()
  {
    timeDisplay.setText(crntTime.toString());
    repaint();
  }

  // handle setting the time
  //
  private void clickedSetTime()
  {
    (new SetTimeDialog(this, crntTime)).show();
  }

  public static void main(String args[])
  {
    (new TimeApp()).show();
  }

  // class local objects
  //
  TextField timeDisplay; // display the time here
  Button getTimeButton;
  Button setTimeButton;
  Button quitButton;
  SysTime crntTime; // representation of the system time
}
```

Note

Note this class is part of the time package.

For More Information

For more information on the Frame class see ***The Java Class Libraries*** and for information on using packages see ***The Java Programming Language***.

The *SetTimeDialog* class

To set the time, use a class that extends `dialog` called `SetTimeDialog` and whose source is located in `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI`. Below is the code implementing this class:

```
package time;
import java.awt.*;
import time.SysTime;
/*
*
* SetTimeDialog
*
*/
public class SetTimeDialog extends Dialog
{
  public SetTimeDialog(Frame parent, SysTime crntTime)
  {
    super(parent, "Set Time Dialog", true);

    resize((6 * (FIELD_WIDTH + 5)) + 40, 100);
    // create the six text fields that we need for the time
    //
    timeFields = new TextField[6];
    timeLabels = new Label[6];
    Integer date[] = crntTime.getTime();
    for (int i = 0; i < 6; i++) {
      timeFields[i] = new TextField(date[i].toString(), 2);
      add(timeFields[i]);
      timeFields[i].reshape(10 + (i * (FIELD_WIDTH + 5 + ((i >= 3) ? 5 : 0))), 24,
                                       FIELD_WIDTH, FIELD_HEIGHT);
      timeLabels[i] = new Label(labels[i]);
      add(timeLabels[i]);
      timeLabels[i].reshape(10 + (i * (FIELD_WIDTH + 5 + ((i >= 3) ? 5 : 0))),
FIELD_HEIGHT
                            + 22, FIELD_WIDTH, FIELD_HEIGHT);
    }
    setButton = new Button("Set");
    add(setButton);
    setButton.reshape(10, (FIELD_HEIGHT * 2) + 22, 50, 20);
    cancelButton = new Button("Cancel");
    add(cancelButton);
    cancelButton.reshape((6 * (FIELD_WIDTH + 5)) - 20, (FIELD_HEIGHT * 2) + 22, 50,
20);
    sysTime = crntTime;
  }

  // handle events for the dialog
  //
  public boolean action(Event event, Object arg)
```

```
  {
    if (event.target == cancelButton) {
      hide();
      return true;
    }
    else if (event.target == setButton) {
      timeVals = new Integer[6];
      for (int i = 0; i < 6; i++) {
if ((timeFields[i].getText()).length() == 0)
  timeVals[i] = new Integer(0);
else
  timeVals[i] = new Integer(timeFields[i].getText());
      }
      sysTime.setTime(timeVals);
      hide();
      return true;
    }
    return false;
  }

  // objects local to this class
  TextField timeFields[];
  Label timeLabels[];
  Integer timeVals[];
  Button setButton;
  Button cancelButton;
  SysTime sysTime;
  private static final int FIELD_WIDTH = 35;
  private static final int FIELD_HEIGHT = 24;
  private static final String labels[] = {"Year", "Mon", "Day", "Hour", "Min",
"Sec"};
}
```

The *SysTime* class

The core of the `TimeApp` application is the `SysTime` class. The source of `SysTime` is located in `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI`. This class is a representation of the system time that supports reading and setting the system time. This class uses two native methods in order to read and set the system time. When first developing the application, these native methods are left out and their behavior is simulated so it is easier to test the Java portion of the application. Below is the source for the `SysTime` class without the native methods:

```
// SysTime - representation of the system time
//
package time;
class SysTime extends Object
{
  // Adding Native Methods - Adding a Static Initialization Block
  // Uncomment this line
  //
  // static {
  // System.loadLibrary("time");
  // }

  // the default constructor
  //
  SysTime()
  {
    // as a default, set the time to an important moment in history
    // Adding Native Methods: Comment out this line
    //
    y = 97; m = 1; d = 15; h = 14; mn = 30; s = 0;
    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line and comment out the line above regarding
    // default assigment to y, m, d, h, mn, and s
    //
    // getSystemTime();
  }

  // construct using an array of Integers
  //
  SysTime(Integer timeVals[])
  {
    if (timeVals.length != 6)
      return;
    y = timeVals[0].intValue();
    m = timeVals[1].intValue();
    d = timeVals[2].intValue();
    h = timeVals[3].intValue();
    mn = timeVals[4].intValue();
    s = timeVals[5].intValue();
```

```
    // Adding Native Methods:  - Calling the Native Methods
    // Uncomment this line
    //
    // setSystemTime();
  }

  //  Adding Native Methods - Adding Declarations
  //  Uncomment these when you are ready to add native method declarations
  //
  // private native void getSystemTime();
  // private native void setSystemTime();

  // return the time as an array of integers
  //
  Integer[] getTime()
  {
    Integer result[] = new Integer[6];

    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line
    //
    // getSystemTime();
    result[0] = new Integer(y);
    result[1] = new Integer(m);
    result[2] = new Integer(d);
    result[3] = new Integer(h);
    result[4] = new Integer(mn);
    result[5] = new Integer(s);
    return result;
  }

  // set the time using an array of Integers
  //
  void setTime(Integer timeVals[])
  {
    if (timeVals.length != 6)
      return;
    y = timeVals[0].intValue();
    m = timeVals[1].intValue();
    d = timeVals[2].intValue();
    h = timeVals[3].intValue();
    mn = timeVals[4].intValue();
    s = timeVals[5].intValue();
    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line
    //
    // setSystemTime();
  }

  // convert the date to a string
  //
  public String toString()
  {
    String  result;
```

```
    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line
    //
    // getSystemTime();
    result = new String(padInt(y % 100) + "/" + padInt(m) + "/" + padInt(d) +
       " " + padInt(h) + ":" + padInt(mn) + ":" + padInt(s));
    return result;
  }

  // pad ints with zeros
  //
  private String padInt(int num)
  {
    String result;

    if (num < 10)
      result = new String("0" + num);
    else
      result = (new Integer(num)).toString();
    return result;
  }

  // objects local to this class
  //
  int     y, m, d, h, mn, s;
}
```

Compiling the Classes

Now that you have read through the examples, compile them using the following command from the `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI` directory:

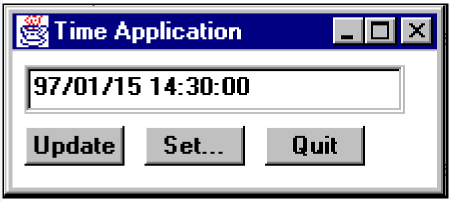
```
javac -d . *.java
```

Running the Example Program

Because `TimeApp` is in the package `time`, the classes for the application need to be in a directory called `time` located on the class search path (specified by either the `CLASSPATH` environment variable or by the `-classpath` command line argument to the Java interpreter). The following is the command line to run the application:

```
java time.TimeApp
```

When run, this application appears as follows on the Windows platform:



When the Set button is clicked in the main application window, an object of the class `SetTimeDialog` is created. When run on the Windows platform this dialog should look similar to the following:

Adding Native Methods

Now that the Java portion of the application is complete, you can implement the native methods and add calls to them in your application.

Step 1: Add Declarations

The first step in this process is to add the method declarations for our native methods. These declarations are added to the `SysTime.java` file and look like the following:

```
private native void getSystemTime();
private native void setSystemTime();
```

These methods receive no parameters since they operate only on the `SysTime` class data.

To add these declarations, search in the `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI\SysTime.java` file for these declarations and uncomment them.

Note

At this point, **recompile** the files as you did earlier so the javah tool can find the native method declarations. With this completed, you can begin to generate the header and stub files necessary to implement the native methods.

Step 2: Generate the Header File

As a part of the PersonalJava Solution for OS-9 package, Microware supplies a special version of the `javah` utility. This utility produces header, stub, and table files used to make a shared library module implementing native methods. Make sure you use the `javah` executable found in `E:\MWOS\DOS\jdk1.1.8\bin`.

Note

If you notice contention with another version of javah, copy the supplied version of javah to the nonJNI directory.

The header file for the `SysTime` class can be generated with the following command line (run from a DOS window):

```
javah -classpath MWOS\DOS\jdk1.1.8\LIB\classes.zip;. time.SysTime
```

This generates a header file called `time_SysTime.h` as follows:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class time_SysTime */

#ifndef _Included_time_SysTime
#define _Included_time_SysTime

typedef struct Classtime_SysTime {
    long y;
    long m;
    long d;
    long h;
    long mn;
    long s;
} Classtime_SysTime;
HandleTo(time_SysTime);

#ifdef __cplusplus
extern “C” {
#endif
extern void time_SysTime_getSystemTime(struct Htime_SysTime *);
extern void time_SysTime_setSystemTime(struct Htime_SysTime *);
#ifdef __cplusplus
}
#endif
#endif
```

Notice the two prototypes generated for the functions `time_SysTime_setSystemTime` and `time_SysTime_getSystemTime`. These are the names of the two C functions you need to write to implement the native methods.

For More Information

Refer to **Step 5: Write the Native Method Functions** on page 100 for instructions on writing these functions.

Step 3: Generate the Stub File

Next, it is necessary to generate the stub `.c` file containing the glue code that takes the Java representation of the `SysTime` object and converts it to a form usable by C functions. This stub file is generated using the following command line:

```
javah -stubs -classpath E:\MWOS\DOS\jdk1.1.8\LIB\classes.zip;. time.SysTime
```

This command generates the file `time_SysTime.c` as follows:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>
#include <slib.h>


/* Stubs for class time_SysTime */
/* SYMBOL: “time_SysTime/getSystemTime()V”, Java_time_SysTime_getSystemTime_stub */
 stack_item *Java_time_SysTime_getSystemTime_stub(stack_item *_P_,struct execenv
*_EE_) {
    extern void time_SysTime_getSystemTime(void *);
    (void) time_SysTime_getSystemTime(_P_[0].p);
    return _P_;
}
/* SYMBOL: “time_SysTime/setSystemTime()V”, Java_time_SysTime_setSystemTime_stub */
 stack_item *Java_time_SysTime_setSystemTime_stub(stack_item *_P_,struct execenv
*_EE_) {
    extern void time_SysTime_setSystemTime(void *);
    (void) time_SysTime_setSystemTime(_P_[0].p);
    return _P_;
}
```

Step 4: Generate the Export Tables

The last source file to create with `javah` contains the table of functions exported by the shared library module. This table is used by the shared library code to perform dynamic runtime linking. To generate the table source file, use the following command line:

```
javah -table -o table.c -classpath E:\MWOS\DOS\jdk1.1.8\LIB\classes.zip;. time.SysTime
```

This command generates the file `table.c` (specified by the `-o` option) as shown below:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <slib.h>
#if defined(NATIVE)
#   include<DEFS/threads.h>
#endif
#include <errno.h>
#include <module.h>

/* these three lines will eliminate the effects of stack checking */
#if     (defined(_MPFPOWERPC) || defined(_MPFARM) || defined(_MPF386) ||
defined(_MPFSH)) && !defined(JAVAMAIN)
void*_stbot = (void *) 0, *_fcbs = (void *) 0;
#if defined(_MPFARM)
_asm("_stkhandler: mov pc,lr"); /* ARM needs a version that doesn't corrupt r11 */
#else
void_stkhandler(void) {}
#endif
u_int32_stklimit = 512*1024;
#endif

externJava_time_SysTime_getSystemTime_stub();
externJava_time_SysTime_setSystemTime_stub();

/* Ptr/Name table for class time_SysTime */
local_function_table_entry sm_local_functions[] = {
    {"Java_time_SysTime_getSystemTime_stub", Java_time_SysTime_getSystemTime_stub},
    {"Java_time_SysTime_setSystemTime_stub", Java_time_SysTime_setSystemTime_stub},
    {NULL, NULL}};
local_ptr_table_entry sm_local_ptrs[] = {
        {NULL, NULL}};
```

Step 5: Write the Native Method Functions

Now that all the necessary header, stub, and table files have been created, you can write the two C functions needed to implement the native methods. For this example, two functions are put in a file called `systime.c` although any name making sense to you can be used. Below is the source for `systime.c`:

```
/*
/*
 * systime.c - native method implementation for the SysTime class
 */

#include        "time_SysTime.h"
#include        <time.h>
#include        <module.h>

/* _sm_bind_main and _sm_unbind_main are pointers to functions.  If these
 * are set to NULL, nothing is done.  If they are initialized to pointers
 * to functions with no parameters, returning void, that function will be
 * called after all module initialization is done in the _sm_bind_main case
 * and before any unbind operations are performed in the _sm_unbind_main case.
 */
void    (*_sm_bind_main)(void) = NULL;
void    (*_sm_unbind_main)(void) = NULL;

        /* time_SysTime_setSystemTime - set system time using class data */

void time_SysTime_setSystemTime(struct Htime_SysTime *this)
{
        Classtime_SysTime       *tptr = unhand(this);
        struct sgtbuf           tbuf;

        tbuf.t_year = tptr->y;
        tbuf.t_month = tptr->m;
        tbuf.t_day = tptr->d;
        tbuf.t_hour = tptr->h;
        tbuf.t_minute = tptr->mn;
        tbuf.t_second = tptr->s;
        if (setime(&tbuf) == -1) {
           SignalError(0, JAVAPKG "InternalError", "error setting system time");
        }
}

        /* SysTime_getSystemTime - use system time to set class data */

void time_SysTime_getSystemTime(struct Htime_SysTime *this)
{
        Classtime_SysTime       *tptr = unhand(this);
        struct sgtbuf           tbuf;
```

```
        if (getime(&tbuf) == -1) {
           SignalError(0, JAVAPKG "InternalError", "error getting system time");
           return;
        }
        tptr->y = tbuf.t_year;
        tptr->m = tbuf.t_month;
        tptr->d = tbuf.t_day;
        tptr->h = tbuf.t_hour;
        tptr->mn = tbuf.t_minute;
        tptr->s = tbuf.t_second;
}
```

Note

The `_sm_bind_main` and `_sm_unbind_main` pointers are set to NULL since this shared library does not require any special initialization. If special setup was required, these pointers would be set to point to the initialization and deinitialization functions.

The `time_SysTime_getSystemTime` and `time_SysTime_setSystemTime` functions use the OS-9 `getime` and `setime` functions respectively to get and set the system time. The fields of the `SysTime` object are referenced by the passed `this` pointers that point to the data members of the `SysTime` object (called `struct Htime_SysTime` in the C functions). All the source files required to implement the native methods are now complete.

Step 6: Compile and Link the Native Method Shared Library

Use Hawk to compile and link the shared library.

For More Information

Refer to ***Using Hawk*** for more information about the following procedures.

Creating a New Project Space and Project

Step 1. To start Hawk on Windows 95/98/Windows NT 4.0, choose `Start-> Programs-> <your OS-9 package> ->Hawk`.

Step 2. Create a new project by selecting `Project -> Project Space->New` from the pull-down menu.

Step 3. Browse to `MWOS\SRC\PJAVA\EXAMPLES\NATIVE\NonJNI` and choose pjava as the project space name. Click `OK`.

Step 4. The **Project Properties** dialog appears. Click `OK` to exit.

Step 5. In the Hawk window choose `Project -> Project Space -> Add New Project`.

Step 6. In the **Create New Project** dialog, enter `libtime` for the project name.

Step 7. Enter `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI` for the project folder.

Step 8. Select the processor from the list.

Example: Generic PowerPC

Step 9. Click `Next` to create a new component for your project.

Creating a New Component

Step 1. Enter `libtime_g` for the component name.

Step 2. Select the processor from the list.

Example: Generic PowerPC

Step 3. Select `User State Program` for the component attributes.

Step 4. Enter `E:\MWOS\OS9000\<proc>\LIB\mt_smstart.r` for the Psect file.

Step 5. Click `Next` to add units to the component.

Adding Units To the Component

Step 1. Enter `systime.c, table.c, time_SysTime.c, time_SysTime.h` for the units of this component.

Step 2. Click `Finish` to set the Project and Component Properties.

Configuring the Project Properties

Step 1. To open the Properties window select `Project->Properties`.

Step 2. Select the `Folders` tab.

Step 3. Enter `E:\MWOS\OS9000\SRC\DEFS\JAVA;MWOS\SRC\DEFS\SPF\BSD;E:\MWOS\SRC\DEFS\UNIX` in the Include field.

Step 4. Select the Source tab.

Step 5. Select Code Generation for Category.

Step 6. Select Source Level for Debug Support.

Step 7. Select Multi-Threading.

Step 8. Click the radio button to enable Multi-threading.

Step 9. Select the options “Thread-safe libraries with fall-back” and “Display warnings given incompatible code”.

Note

If you are not using the debugging version of the shared library (you are using libtime.so instead of libtime_g.so), you must set Debug Support to None instead of Source Level. If you choose this method, you will not be able to debug your code using Hawk.

Step 10. Select the Link tab.

Step 11. Select General for Category.

Step 12. Enter `libbinding.l,sys_clib.l,libsm.l,cpu.l` into the O-Code Library field. Even though all these appear to be non-threading libraries, the compiler will automatically choose the threading version of these libraries if available.

Note

Order is important in this field.

Step 13. Select Customization for Category.

Step 14. Enter `E:\MWOS\OS9000\<proc>\LIB\mt_smstart.r` into Psect.

Note

Step 11 was optional, though recommended. If you decide to add components later, the Psect will already be defined.

Step 15. Select the Debug tab.

Step 16. Enter the target name or target IP address into the Address field.

Step 17. Click Close.

Specifying Component Properties

Step 1. With your mouse, right click on the `libtime_g` component.

Step 2. Click on properties.

Step 3. Select the General tab.

Step 4. Change the output file name to `libtime_g.so`.

Step 5. Click Close.

Step 6. To save your project, select `Project->Save`.

Step 7: Call the Native Methods from the *SysTime* Class

Use the Editor in Hawk to add the calls to the native methods to the `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI\SysTime.java` file. The calls you need to add are highlighted in the source file below:

```
// SysTime - representation of the system time
//
package time;
class SysTime extends Object
{
  // Adding Native Methods - Adding a Static Initialization Block
  // Uncomment this line
  //
 // static {
 // System.loadLibrary("time");
 // }

  // the default constructor
  //
  SysTime()
```

```
{
  // as a default, set the time to an important moment in history
  // Adding Native Methods: Comment out this line
  //
  y = 97; m = 1; d = 15; h = 14; mn = 30; s = 0;
  // Adding Native Methods - Calling the Native Methods
  // Uncomment this line and comment out the line above regarding
  // default assignment to y, m, d, h, mn, and s
  // getSystemTime();
}

// construct using an array of Integers
//
SysTime(Integer timeVals[])
{
  if (timeVals.length != 6)
    return;
  y = timeVals[0].intValue();
  m = timeVals[1].intValue();
  d = timeVals[2].intValue();
  h = timeVals[3].intValue();
  mn = timeVals[4].intValue();
  s = timeVals[5].intValue();

  // Adding Native Methods:  - Calling the Native Methods
  // Uncomment this line
  //
 // setSystemTime();
}

//  Adding Native Methods - Adding Declarations
//  Uncomment these when you are ready to add native method declarations
//
   private native void getSystemTime();
   private native void setSystemTime();

// return the time as an array of integers
//
Integer[] getTime()
{
  Integer result[] = new Integer[6];

  // Adding Native Methods - Calling the Native Methods
  // Uncomment this line
  //
 // getSystemTime();
  result[0] = new Integer(y);
  result[1] = new Integer(m);
  result[2] = new Integer(d);
  result[3] = new Integer(h);
  result[4] = new Integer(mn);
  result[5] = new Integer(s);
  return result;
}
```

```
  // set the time using an array of Integers
  //
  void setTime(Integer timeVals[])
  {
    if (timeVals.length != 6)
      return;
    y = timeVals[0].intValue();
    m = timeVals[1].intValue();
    d = timeVals[2].intValue();
    h = timeVals[3].intValue();
    mn = timeVals[4].intValue();
    s = timeVals[5].intValue();
    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line
    //
   // setSystemTime();
  }

  // convert the date to a string
  //
  public String toString()
  {
    String  result;

    // Adding Native Methods - Calling the Native Methods
    // Uncomment this line
    //
   // getSystemTime();
    result = new String((y) + "/" + padInt(m) + "/" + padInt(d) + " " + padInt(h) +
":
                        " + padInt(mn) + ":" + padInt(s));
    return result;
  }

  // pad ints with zeros
  //
  private String padInt(int num)
  {
    String result;

    if (num < 10)
      result = new String("0" + num);
    else
      result = (new Integer(num)).toString();
    return result;
  }

  // objects local to this class
  //
  int     y, m, d, h, mn, s;
}
```

Step 8: Add Calls to the Native Methods

To add calls to the native methods from `SysTime` complete the steps below:

Step 1. Search in `SysTime.java` for `Calling the Native Methods` and uncomment the line `getSystemTime()`.

Step 2. Continue searching in `SysTime.java` for `Calling the Native Methods` and uncomment the line `setSystemTime()`.

Step 3. Continue searching in `SysTime.java` for `Calling the Native Methods` and uncomment the line `getSystemTime()`.

Step 4. Continue searching in `SysTime.java` for `Calling the Native Methods` and uncomment the line `setSystemTime()`.

Step 5. Continue searching in `SysTime.java` for `Calling the Native Methods` and uncomment the line `getSystemTime()`.

Step 9: Add a Static Initialization Block to Load the Shared Library

The final addition needed to the `SysTime.java` file is the static initialization block. The static initialization block for the `SysTime` is called only once so this is a convenient place to load the shared library. The code for this block is as follows:

```
static {
   System.loadLibrary("time");
}
```

The library name passed to `System.loadLibrary` is `time`. When the Java interpreter attempts to load the library, it prepends `lib` to the name and appends either `_g.so` or `.so` depending on whether the `pjava_g` (debugging) or `pjava` interpreter is being used.

To add the static initialization block, search in `systime.java` for `Adding a Static Initialization Block` and uncomment these lines.

At this point, recompile the java classes to generate the enhanced `.class` files.

Step 10: Compiling and Linking

From Hawk, select `Project->Build` to build the component.

Ignore the warning about `libbinding.l` having a thread incompatibility. It is a completely thread-safe library.

Three files are generated in the directory: `libtime_g.so`, `libtime_g.so.dbg`, and `libtime_g.so.stb`.

The shared library is now complete and ready to test.

Running the *TimeApp* Application on the Target

To run the TimeApp application on your target, complete the following steps:

Step 1: Transfer the Class Files

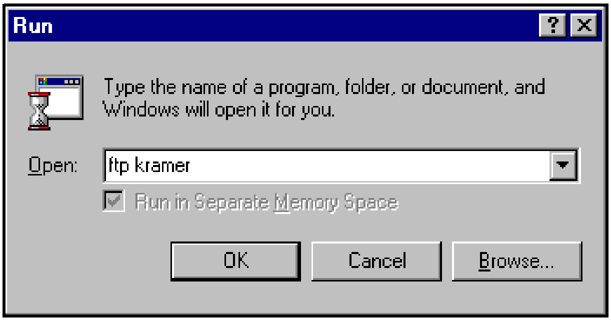
Note

These steps assume that you have a system disk on your target machine. Use these steps as a guide for transferring the class files and shared libraries to the OS-9 target machine.

Now that the application is complete and compiled (including the shared library), it is time to transfer the class and shared library code to the target OS-9 system. This is done in this example using FTP. This could also be done using NFS. The steps to transfer using FTP are as follows:

Step 1. Choose `Start->Run` on the Windows desktop.

Step 2. In the Run dialog box, type ftp <machine name> then click OK. The machine name in this example is kramer.



Step 3. Log on to the OS-9 machine by typing the user name and password in the FTP (MS-DOS Shell) window. The default user name and password for OS-9 machines is super and user.

Step 4. Change to the directory containing the application classes on the windows machine by typing the following in the FTP window:

```
lcd E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI\time
```

Step 5. Change to the demo directory on the OS-9 machine by typing the following in the FTP window:

```
cd /h0/MWOS/SRC/PJAVA
```

Step 6. Create the time directory on the OS-9 machine by typing the following FTP commands:

```
mkdir EXAMPLES
mkdir EXAMPLES/NATIVE
mkdir EXAMPLES/NATIVE/nonJNI
mkdir EXAMPLES/NATIVE/nonJNI/time
```

Step 7. Change to the time directory on the OS-9 machine by typing the following in the FTP window:

```
cd EXAMPLES/NATIVE/nonJNI/time
```

Step 8. Change to binary transfer by typing the following in the FTP window:

```
bin
```

Step 9. Turn off FTP interactive mode by typing the following in the FTP window:

```
prompt
```

Step 10. Transfer the class files by typing the following in the FTP window:

```
mput *.class
```

Now transfer the shared library to the OS-9 system.

Step 11. Move up one directory on the Windows machine by typing the following in the FTP window:

```
lcd ..
```

Step 12. Change to the shared library directory on the OS-9 system by typing the following in the FTP window:

```
cd /h0/MWOS/OS9000/<proc>/LIB/SHARED
```

Step 13. Transfer the shared libraries to the OS-9 system by typing the following in the FTP window:

```
mput libtime_g*
```

Step 14. Quit the FTP session.

Now that all the object code needed to run the application has been transferred to the OS-9 machine, you are ready to test and debug the application.

Step 2: Start the Java Application on the OS-9 System

In the below example, telnet is used to communicate with the OS-9 system.

Starting Telnet Session

To start a telnet session perform the following steps:

Step 1. Choose `Start->Run` from the Windows desktop.

Step 2. In the Run window text field enter

```
telnet <target system>
```

and click OK.

Step 3. Log onto the OS-9 system by typing the user name and password. `Super` and `User` are the defaults for OS-9 systems.

Note

Before running the application, the permissions must be correctly set on the shared library module so it can be loaded by the OS-9 system at run time.

In the telnet window, type the following command:

```
chd /h0/MWOS/OS9000/<proc>/LIB/SHARED
attr -pegeeprgrr libtime_g*
```

This sets the public execute and public read permissions for all the libtime_g files previously transferred from the Windows machine.

Setting Variables

In order for the Java interpreter to find the class files and shared libraries, certain environment variables must be set. Follow these steps to set the variables:

Step 1. Set the `CLASSPATH` environment variable so the interpreter can find the classes for the `TimeApp` application by typing the following command in the telnet window:

```
setenv CLASSPATH /h0/MWOS/SRC/PJAVA/EXAMPLES/NATIVE/nonJNI:$CLASSPATH
```

Step 2. Set the `LD_LIBRARY_PATH` environment variable with this command:

```
setenv LD_LIBRARY_PATH /h0/MWOS/OS9000/<proc>/LIB/SHARED:$LD_LIBRARY_PATH
```

Step 3. Run the application by typing the following command in the telnet window:

```
pjava_g time.TimeApp &
```

Note

Be sure that you are using pjava_g. This is the debugging version of the PJava virtual machine.

The `TimeApp` application window now appears on the graphic device connected to the target OS-9 system. Try setting and updating the time, or entering `date` into the console, to make sure the application is working.

Debugging Native Methods

The next steps in this example deal with using the Hawk debugger to examine the behavior of the code in the native method shared library.

Debugging with Hawk

Follow the steps below to debug with Hawk:

Step 1. Reboot the target.

Step 2. Make sure the Hawk SPF daemon is running before proceeding by typing the `procs` command with the `-e` option as follows:

```
$ procs -e
 Id PId Grp.Usr  Prior  MemSiz Sig S    CPU Time   Age Module & I/O
  2   0   0.0   128 24.00k  0 s  0.00  2:00 spfndpd <>>>nil <--SPF daemon
  3  12   0.88  128 56.00k  0 *  0.04  0:00 procs <>>>pks01
```

If the daemon is not running, complete the following on your OS-9 target at the OS-9 prompt:

```
$ load -d /h0/CMDS/spfndpd
$ load -d /h0/CMDS/spfndpdc
$ spfndpd <>>>/nil &
```

Identifying Source and Object Code

Step 1. Start Hawk.

Step 2. Select `Debug->Option->Folders`.

Step 3. Enter `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI` in the Source Code field.

Enter `E:\MWOS\OS9000\<proc>\CMDS` and `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\nonJNI` in the Object Code field.

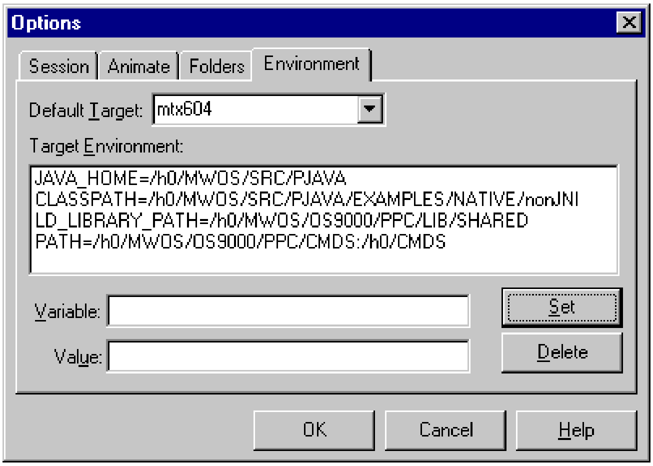
Step 4. Click OK

Setting Up Hawk Target Environment

Step 1. Select `Debug->Option->Environment`.

Step 2. Set the values for `CLASSPATH`, and `JAVA_HOME`, `PATH`, and `LD_LIBRARY_PATH` variables to the following:

```
CLASSPATH=/h0/MWOS/SRC/PJAVA/EXAMPLES/NATIVE/nonJNI
JAVA_HOME=/h0/MWOS/SRC/PJAVA
PATH=/h0/MWOS/OS9000/<proc>/CMDS:/h0/CMDS
LD_LIBRARY_PATH=/h0/MWOS/OS9000/<proc>/LIB/SHARED
```



Forking the Java Process

Fork the Java process by completing the steps below:

Step 1. Choose `Debug->Connect`.

Step 2. Enter the target name.

Step 3. Enter `pjava_g` in the Program text field in the Fork dialog box.

Step 4. Enter `time.TimeApp` in the Parameters text field.

Step 5. Notice the values in the Target Environment Variables area.

Step 6. Click `OK` in the Fork dialog box.

Note

The debugger displays warning dialog boxes because it cannot find debugging information for the pjava_g module. This is not a problem and the dialog boxes can be dismissed by clicking OK.

Loading the Shared Library

Before attaching to the shared library, it must be loaded into memory. From the OS-9 prompt on the target system enter the following:

```
load -ld /h0/MWOS/OS9000/<proc>/LIB/SHARED/libtime_g.so
```

Linking to the Shared Library

At this point the search paths have been set. Use the following steps to link to the `libtime_g.so` shared library:

Step 1. Choose `Debug->Connect` from the Debugger menu.

Step 2. Select the Attach tab.

Step 3. Choose System for the type.

Step 4. Enter the target name.

Step 5. Choose Module.

Step 6. Enter `libtime_g.so` in the Module field.

Step 7. Click `OK`.

Note

You may receive a warning that Hawk cannot open a path to the server. This is not a problem. Ignore the warning by clicking OK.

Setting Breakpoints

To set breakpoints on the Native Method functions, complete the steps below:

Step 1. Chose `Debug->View->Browse Symbol...` from the debugger menu.

Step 2. Click on the `+` symbol for `libtime_g.so` in the browser window to expand all the symbols in the shared module.

Step 3. Click on the `+` symbol for `systime.c` to see all the functions in the source code.

Step 4. Right click on the function `time_SysTime_getSystemTime`.

Step 5. Choose `Toggle Breakpoint` in the browser window.

Step 6. Right click on the function `time_SysTime_setSystemTime`.

Step 7. Choose `Toggle Breakpoint` in the browser window.

Step 8. Close the browser window.

Step 9. Click on the `Run` button in the debugger's toolbar to run the Java process.

The Java interpreter stops when the break points are hit. To run the interpreter again, click on `Run` until the Time Application is displayed.

For More Information

For information about debugging on OS-9 systems see ***Using Hawk***.

Using JNI Native Methods

Introduction to JNI Native Methods

With the 1.1 version of the Java Development Kit, a new native method calling mechanism called Java Native Interface was added.

The JNI specification describes the advantage of using JNI as follows:

“The most important benefit of the JNI is it imposes no restrictions on the implementation of the underlying JVM. Therefore, JVM vendors can add the support for the JNI without affecting other parts of the JVM. Programmers can write one version of native application or library and expect it to work with all JVMs supporting the JNI.”

For More Information

The specification for the JNI can be found at the following location:

http://java.sun.com/

This example shows how to implement the TimeApp application using JNI Native Methods in place of the older Native Method mechanism. Only the steps that differ from those described for the first pass through this tutorial are described here. Otherwise follow the same process, except where these differences are noted.

Note

While completing the JNI Native Methods example, insert JNI in the place of nonJNI wherever it appears in the directory listings in the tutorial.

Generating the JNI Header Files

The same `javah` tool used to generate the header files for the previous example is used to generate header files for JNI Native Methods. The command line that generates the JNI header file (run from a MS-DOS window with the current directory set to `E:\MWOS\SRC\PJAVA\EXAMPLES\NATIVE\JNI`) for the `SysTime` class is shown below:

```
javah -jni -classpath E:\MWOS\DOS\jdk1.1.8\LIB\classes.zip;. time.SysTime
```

This generates a header file called `time_SysTime.h` as follows:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class time_SysTime */

#ifndef _Included_time_SysTime
#define _Included_time_SysTime
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     time_SysTime
 * Method:    getSystemTime
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_time_SysTime_getSystemTime
  (JNIEnv *, jobject);

/*
 * Class:     time_SysTime
 * Method:    setSystemTime
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_time_SysTime_setSystemTime
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Note the two function prototypes for the functions `Java_time_SysTime_setSystemTime` and `Java_time_SysTime_getSystemTime`. These are the functions you need to write. The source for these functions is provided in the following section.

Generating the JNI Stub File

Due to the nature of the JNI, no stub functions are needed to interface the JVM to JNI native methods. The JVM is able to call them directly.

Generating the JNI Export Tables

To generate the JNI export table source file, use the following command line:

```
javah -jni -table -o table.c -classpath E:\MWOS\DOS\jdk1.1.8\LIB\classes.zip;.
time.SysTime
```

This command generates the file `table.c` (specified by the `-o` option) as follows:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <slib.h>
#if defined(NATIVE)
#   include<DEFS/threads.h>
#endif
#include <errno.h>
#include <module.h>

/* these three lines will eliminate the effects of stack checking */
#if     (defined(_MPFPOWERPC) || defined(_MPFARM) || defined(_MPF386) ||
defined(_MPFSH)) && !defined(JAVAMAIN)
void*_stbot = (void *) 0, *_fcbs = (void *) 0;
#if defined(_MPFARM)
_asm("_stkhandler: mov pc,lr"); /* ARM needs a version that doesn't corrupt r11 */
#else
void_stkhandler(void) {}
#endif
u_int32_stklimit = 512*1024;
#endif

externJava_time_SysTime_getSystemTime();
externJava_time_SysTime_setSystemTime();

/* Ptr/Name table for class time_SysTime */
local_function_table_entry sm_local_functions[] = {
    {"Java_time_SysTime_getSystemTime", Java_time_SysTime_getSystemTime},
    {"Java_time_SysTime_setSystemTime", Java_time_SysTime_setSystemTime},
    {NULL, NULL}};

local_ptr_table_entry sm_local_ptrs[] = {
        {NULL, NULL}};
```

Writing the JNI Native Method Functions

Below is the source for the JNI version of `systime.c`.

```
/*
 * systime.c - native method implementation for the SysTime class
 */

#include        "time_SysTime.h"
#include        <time.h>
#include        <module.h>

/* _sm_bind_main and _sm_unbind_main are pointers to functions.  If these
 * are set to NULL, nothing is done.  If they are initialized to pointers
 * to functions with no parameters, returning void, that function will be
 * called after all module initialization is done in the _sm_bind_main case
 * and before any unbind operations are performed in the _sm_unbind_main case.
 */
void    (*_sm_bind_main)(void) = NULL;
void    (*_sm_unbind_main)(void) = NULL;

        /* time_SysTime_setSystemTime - set system time using class data
         */
void Java_time_SysTime_setSystemTime(JNIEnv *envptr, jobject obj)
{
        struct sgtbuf           tbuf;
        jclass                  clazz;
        JNIEnv                  env = *envptr;

        clazz = env->GetObjectClass(envptr, obj);

        tbuf.t_year = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "y", "I"));
        tbuf.t_month = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "m", "I"));
        tbuf.t_day = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "d", "I"));
        tbuf.t_hour = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "h", "I"));
        tbuf.t_minute = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "mn", "I"));
        tbuf.t_second = env->GetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "s", "I"));
        if (setime(&tbuf) == -1) {
                env->ThrowNew(envptr, env->FindClass(envptr,
                        "java/lang/InternalError"),"error setting system time");
        }
}

        /* SysTime_getSystemTime - use system time to set class data
         */
void Java_time_SysTime_getSystemTime(JNIEnv *envptr, jobject obj)
```

```
{
        struct sgtbuf           tbuf;
        jclass                  clazz;
        JNIEnv                  env = *envptr;

        clazz = env->GetObjectClass(envptr, obj);
        if (getime(&tbuf) == -1) {
                env->ThrowNew(envptr, env->FindClass(envptr,
                        "java/lang/InternalError"),"error getting system time");
                return;
        }

        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "y", "I"), tbuf.t_year);
        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "m", "I"), tbuf.t_month);
        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "d", "I"), tbuf.t_day);
        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "h", "I"), tbuf.t_hour);
        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "mn", "I"), tbuf.t_minute);
        env->SetIntField(envptr, obj,
                env->GetFieldID(envptr, clazz, "s", "I"), tbuf.t_second);
}
```

Compiling and Linking the JNI Native Method Shared Library

Now that the source files are ready, use Hawk again to build the component.

Follow the directions found in **Step 6: Compile and Link the Native Method Shared Library** on page 102. The only difference is in the section **Adding Units To the Component** on page 103. Only the files `systime.c` and `table.c` need to be added to the project for the JNI example.

As in the previous example, three files are generated in the current directory: `libtime_g.so`, `libtime_g.so.dbg`, and `libtime_g.so.stb`.

The JNI shared library is now complete and ready to test. Refer to the section **Running the TimeApp Application on the Target** on page 110 for instructions on transferring the class files and shared library to the test machine and subsequent debugging of the application.

Chapter 7: Using the Window Manager

The window manager is a MAUI (Multimedia Application User Interface) application that must be running before you run any Java applications or applets that display graphics. This chapter describes how to use the window manager. It includes the following topics:

Note

These examples use the `E:\` directory. The location of your `MWOS` tree may vary depending on where you chose to install your PersonalJava Solution for OS-9 package).

Window Manager Process

To determine if the window manager is running, type `procs -e` from the command line on your target. You should see a listing similar to the following:

```
$ procs -e
 Id PId Grp.Usr  Prior  MemSiz Sig S    CPU Time   Age Module & I/O
  2   0   0.148   128   48.50k   0 w        0.10  0:00 mshell <>>>term
  3   0   0.0     128   66.50k   0 s        0.00  0:01 telnetd <>>>nil
  4   0   0.0     128   16.25k   0 e        0.02  0:01 spf_rx
  5   0   0.0     128   50.75k   0 s        0.01  0:01 ftpd <>>>nil
  6   2   0.148   128   11.50k   0 s        0.01  0:00 spfndpd <>>>nil
  7   2   0.148   128   48.00k   0 *        0.00  0:00 procs <>>>term
  8   0   0.148   256   26.75k   0 s        0.01  0:00 maui_inp <>>>nil
  9   0   0.148   250  159.25k   0 s        0.58  0:00 winmgr <>>>nil
 10   9   0.148   256   47.25k   0 s        0.46  0:00 maui_win <>>>nil
```

The process noted by process ID 9 in the listing above indicates the window manager is running.

For More Information

For an example of forking the window manager, refer to Running the Demos in the ***Getting Started With PersonalJava Solution for OS-9*** manual.

Window Managers

There are three window managers for the PersonalJava Solution for OS-9 installation. To choose one, read the following descriptions:

Simple Window Manager

The base window manager performs the following:

- Parses the settings file
- Opens devices designated in the Configuration Description Block
- Allocates colors designated in settings file
- Assigns arrow cursor to root window
- Sets system keys designated in settings file

Standard Window Manager

The standard window manager performs the following:

- performs the Simple Window Manager activities
- reparents client application windows
- drags/resizes client application windows
- monitors client applications
- uses cursors and icons from Resource module
- uses a root menu (activated by button 3 click on root window) with the following options:
  - refresh root window
  - refresh all
  - shutdown
  - arrange icons

- lowers/raises client application windows
- draws the frame, title bar, and other title bar items for client application windows

  The buttons on the title bar include (left-to-right):

  •kill the application

  •window management options pull-down menu that includes:

    •move

    •resize

    •raise

    •lower

    •minimize

    •maximize

  •minimize the client application window to an icon

  •maximize/Minimize the client application

Debugging Window Manager

The debugging window manager performs the following:

- performs the Standard Window Manager activities
- adds the following options to the root menu:

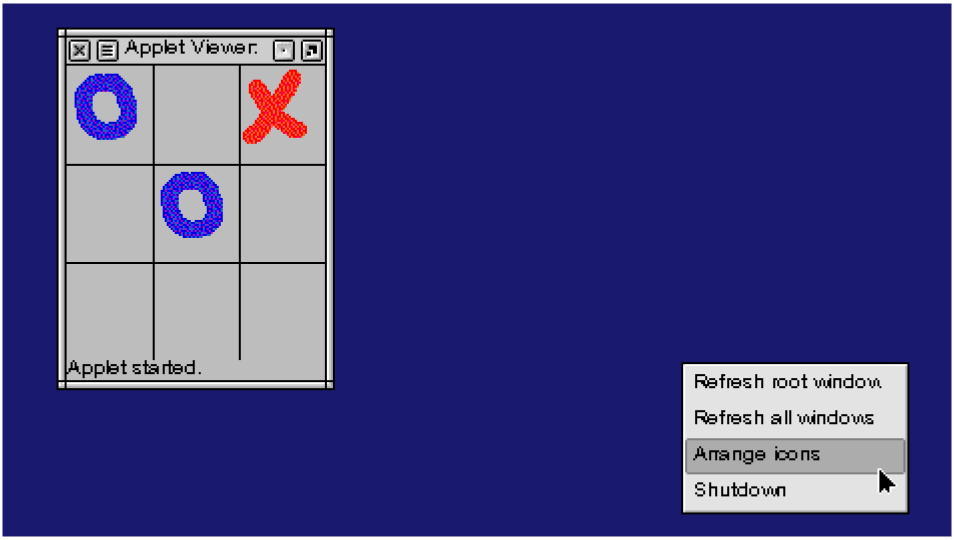
  •send shutdown message

  •dump window tree

- prints memory usage information upon termination if the `MEMWATCH` environment variable is set

Sample Window Manager

Below is an example of the Standard Window Manager:



Using the Window Manager

The window manager is command-line driven using the command below:

Note

The window manager must have a higher priority than any client application. Any application that creates, destroys, moves, or resizes a window frame frequently can cause the window manager to get behind. This may result in the system failing. When forking the window manager, use `winmgr ^250 &`.

Command

```
winmgr [&lt;args&gt;]
```

Arguments

`-font=<fontName>` specifies the font file name

`-fontsize=<fontSize>`
specifies the size of the font in points

`-fontfamily=<family>`
specifies the below font family:

- `MT` is Micro Type (default).
- `MTPLUG` is a set of glyphs common to a number of fonts (and includes items such as squares and arrows). These items do not change with the rest of the type face.
- TT is True Type.
- PS is postscript.

`-focusPointer [-focusRaise] [-focusButton]`
specifies how the window gets focus.

- `focusPointer`: The window gets focus when the mouse crosses the window border.
- `focusRaise`: The window gets focus when the mouse crosses the window border and the window is brought to the front of the other windows.
- `focusButton`: The window gets focus when the mouse clicks on the window.

`-gfxdev=<graphicDevice>`
: specifies the graphic device id

`-windev=<winDevice>`
: specifies the window device id

`-dev=<settingsDevName>`
: specifies which settings to use in the resource file

The default is `dev1`. Refer to the **Example Preference File** on page 134 in this chapter.

`-ntsc` specifies an NTSC display

`-queue=<minimum queueSize>]`
: specifies the number of messages the mailbox can hold

`-settings=<settingsModule>]`
: specifies the name of the window manager resource file/module

Window Manager Preference File

Example Preference File

Below is an example window manager preference file:

```
#
# Sample settings file for AFW window managers
#

#
# winmgr application settings
#
queuesize=300              # Suggested minimum queue size for use with Java
frameicon=0                # Determines if icons are framed (0 = noframe, 1 = frame)

winmgrfont=mw_java.fco,MT  # Font used by window manager for frame titles, icon labeling
                           # (Micro Type fonts only through this mechanism: for PS or TT
                           # fonts, use command-line specification and appropriate flags)

#
# Numeric setting controls
#
poll=10000                 # Interval between client app "pings", in MS
clickTime=40               # Maximum interval for multi-click, in MS
focus=2                    # Focus policy (0 = pointer, 1 = pointer&raise, 2 = button)

#
# Colors are now Device specific
# dev1 is the default device
#
dev1.resIndex=0
dev1.cmIndex=0
dev1.white=255,255,255     # RGB value for stock white color
dev1.black=0,0,0           # RGB value for stock black color
dev1.light=101,142,220     # RGB value for stock light color
dev1.dark=82,116,179       # RGB value for stock dark color
dev1.grey1=227,227,227     # RGB value for stock grey1 color
dev1.grey2=205,205,205     # RGB value for stock grey2 color
dev1.grey3=189,189,189     # RGB value for stock grey3 color
dev1.grey4=172,172,172     # RGB value for stock grey4 color
dev1.grey5=156,156,156     # RGB value for stock grey5 color
dev1.grey6=128,128,128     # RGB value for stock grey6 color
dev1.screen=25,25,112      # RGB value for screen background color (midnight blue)

#
# Various key bindings
#   - Key specification may be by either quoted characters (e.g., 'x')
#     or by MAUI keycode values (hex numbers only, please)
#   - Supported modifiers are ALT, CTRL, and SHIFT
#

SysMenuKey=' ' + ALT       # Activates system menu
RootMenuKey='+'+ALT+CTRL+SHIFT # Activates root window menu (not fully supported)
SwitchKey=0x9 + ALT        # Switches between windows (future use)

LeftKey=0xff51             # Binding for signaling "right" direction on frames
RightKey=0xff53            # Binding for signaling "left" direction on frames
UpKey=0xff52               # Binding for signaling "up" direction on frames
```

```
DownKey=0xff54             # Binding for signaling "down" direction on frames

AcceptKey=0x0d             # Binding for signaling acceptance of position/size
CancelKey=0x1b             # Binding for canceling reposition/resize

CutKey='x' + CTRL          # Binding for 'cut' operations
CopyKey='c' + CTRL         # Binding for 'copy' operations
PasteKey='p' + CTRL        # Binding for 'paste' operations
```

Preference File Location

The window manager preferences are located in a data module named `winmgr.dat`, on your Windows host machine in the `E:\MWOS/OS9000/<target>/CMDS` directory.

Disk-based System

If your target has a system disk, the preferences data module `winmgr.dat` will be loaded by the `loadjava` script from `E:\MWOS/OS9000/<target>/CMDS`.

Diskless System

If your target does not have a hard drive, the `winmgr.dat` data module should be added to the list of modules that is transferred to the OS-9 target machine.

Note

A text version of the Window Manager preference file is in the directory `E:\MWOS\SRC\AFW\WINMGR` on the Windows host machine. The name of the text file is `winmgr.txt.`

Editing the Preference File

If you want to edit the preference file for your target requirements, you can modify any of the lines in this file to values supported by your target. To edit the preference file, complete the following steps:

Step 1. On the Windows host machine, change to the directory where the Window Manager preference file is located by typing the following:

```
cd MWOS\SRC\AFW\WINMGR
```

Step 2. Copy the text version of the Window Manager preference file as a back up of its original state.

```
copy winmgr.txt winmgr.org
```

Note

The copy of the text version of the Window Manager preference file, `winmgr.org`, has DOS line endings.

Step 3. Use the Windows editor of your choice to edit and save the file `winmgr.txt`.

Step 4. Use the OS-9 utility `cudo` on your Windows machine to change the line endings from DOS to OS-9 in the file `winmgr.txt` by typing the following:

```
cudo -cdo winmgr.txt
```

Step 5. Use the OS-9 utility `mkdatmod` on your Windows machine to package the file `winmgr.txt` into an OS-9 data module by typing the following:

```
mkdatmod winmgr.txt -to=os9000 -tp=<port_proc> winmgr.dat -n=winmgr.dat
```

From this point, if you are using a disk-based system, complete steps six and seven. However, if you are using a diskless system, skip steps six and seven and complete step eight.

Disk-based System

Step 6. Transfer your Window Manager preferences data module (`winmgr.dat`) from the Windows host machine to the `/h0/MWOS/OS9000/<proc>/CMDS` directory on the OS-9 target machine. It should replace the Window Manager preferences data module that was created when you installed the PersonalJava Solution for OS-9 on your target machine. Your data module should be transferred to the OS-9 target machine in binary mode.

Step 7. Reboot the OS-9 target machine. Executing the loadjava script loads your data module (`winmgr.dat`) from the directory `/h0/MWOS/OS9000/<proc>/CMDS` into the module directory.

Diskless System

Step 8. Copy your new Window Manager preferences data module to `E:\MWOS\OS9000\<proc>\CMDS` on the Windows host machine. The new Window Manager preferences data module is then ready to be included in the next build of pjruntime.

Running MAUI Applications with PersonalJava Applications

Standard MAUI applications can be executed at the same time as graphical PersonalJava applications as long as the MAUI windowing API (Application Programming Interface) is used to access the screen. PersonalJava Solution’s window manager only pays attention to PersonalJava applications. Other MAUI applications can be executed and will overlap suitably with PersonalJava applications.

Window Manager Error Codes

Table 7-1 lists and defines all possible Window Manager errors codes.

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:000	`EAFW_NOTIMPLEMENTED`	
012:001	`EAFW_NOTDEFINED`	No drawing method defined at object instantiation.
012:002	`EAFW_DEFINED`	Object already defined. This pertains to the application or color manager object. There can only be one of each of these per process.
012:003	`EAFW_BADPOS`	Either the scroll position or the insertion position is not valid for the object.
012:004	`EAFW_BADPENCOUNT`	The number of pens set must be 0 if pixels is NULL. Or, if pixels is not NULL, the number of pens set must be greater than 0.
012:005	`EAFW_BADCM`	Not used.

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:006	`EAFW_INUSE`	This error can mean one of three things: 1) An attempt was made to register a MAUI message converter function to an event that already has a converter function registered. 2) The same signal is monitored in more than one handler. 3) An application is either re-opening a MAUI window device or it is attempting to open another window device. (The AFW is designed to allow only one window device per application.)

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:007	EAFW_NOTFOUND	This error can mean one of three things: 1) The AGFA font module is not present in the module directory. 2) An attempt was made to ignore an unknown signal. 3) In a text object, this error means there has been a search through the text buffer, an occurrence of the specified text starting at the specified position in the buffer.
012:008	EAFW_BADMSGTYPE	An attempt was made to register a message type that has no MAUI message converter function specified.
012:009	EAFW_QEMPTY	Not used.
012:010	EAFW_NOGFXDEV	No graphics device name was specified for the owner window device instantiation (`MOwnerWinDev`).

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:011	`EAFW_NOWINDEV`	The application is unable to access the MAUI window device--it was either not instantiated or it has been destroyed for some reason. The MAUI windowing device name must be specified for the owner window device instantiation (`MOwnerWinDev`).
012:012	`EAFW_NOCOLORMGR`	One color manager (`MColorManager`) must be defined to draw anything. This manager should be defined after the windowing device is instantiated. This error occurs when the color manager is not defined.
012:013	`EAFW_NOAPP`	Exactly one application object (`MApplication`) can be instantiated. For some reason, the application can not be found.

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:014	`EAFW_BADWINID`	The MAUI window ID of the child window is invalid. (The MAUI window being framed is called the child window or client window.) This can also occur when an attempt is made to find the active menu, when no menu is defined or when a GUI widget is realized before a root window is initialized.
012:015	`EAFW_BADMODULE`	MAUI text is instantiated with a bad font module name.

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:016	EAFW_BADCOLOR	The color encoding type specified upon creation of the window device (MOwnerWinDev) was GFX_COLOR_NONE. The options are listed below: GFX_COLOR_NONE: No color encoding GFX_COLOR_RGB: RGB color(s) GFX_COLOR_YUV: YUV color(s) GFX_COLOR_A1_RGB: RGB with alpha flag GFX_COLOR_YCBCR: YCbCr color(s) GFX_COLOR_1A7_RGB: RGBalpha flag & value GFX_COLOR_1A7_YCBCR: YCbCr alpha flag & value GFX_COLOR_A8_RGB: RGB with 8-bit alpha value
012:017	EAFW_BADCONTEXT	Not used.
012:018	EAFW_MEMFAIL	Not used.
012:019	EAFW_BADFONT	Not used.
012:020	EAFW_BADTEXTCONTEXT	Not used.

Table 7-1 Window Manager Error Codes

Error Code	Name	Definition
012:021	EAFW_NOTINITIALIZED	Not used.
012:022	EAFW_BADPARAM	Not used.
012:023	EAFW_BADRESOURCE	Not used.
012:024	EAFW_NODEVICES	Not used.
012:025	EAFW_BADSIZE	Not used.
012:026	EAFW_BADATTRIBS	Not used.
012:027	EAFW_INVALIDOP	Not used.
012:028	EAFW_BADSTYLE	Not used.
012:029	EAFW_NOCREATECHILD	Not used.
012:030	EAFW_NULL	Not used.
012:031	EAFW_AWFNULL	Not used.
012:032	EAFW_DAWFNULL	Not used.
012:033	EAFW_DAWOFNULL	Not used.
012:034	EAFW_DEFSIZE	Not used.
012:035	EAFW_IMAGENULL	Not used.
012:036	EAFW_MBOXLONGNAME	Not used.
012:037	EAFW_INTERNALERR	Not used.

Chapter 8: Enhancing the Properties Files

PersonalJava Solution for OS-9 offers several ways to customize how the JVM (Java Virtual Machine) runs on the OS-9 target machine. This chapter provides information about the Microtype fonts provided with this version of PersonalJava Solution for OS-9, and instructions for modifying `font.properties` and `awt.properties` to change the behavior of the JVM on the OS-9 target machine.

This chapter includes the following topics:

Microtype Fonts

The Microtype fonts can be found on your Windows development host at `E:\MWOS\OS9000\<target>\ASSETS\FONTS\AGFA` and your OS-9 target machine at `/h0/MWOS/OS9000/<proc>/ASSETS/FONTS/ AGFA`. The modules include the following:

`mw_java.fco`	PersonalJava Solution for OS-9 font data module
`mwp_java.fco`	PersonalJava Solution for OS-9 font plug-in module.
`mt.ss`	PersonalJava Solution for OS-9 font symbol set data module

Table 8-1 describes the fonts included in PersonalJava Solution for OS-9:

Table 8-1 Microtype Fonts

Family Name	Style	Index Entry For the Family
Serif	plain	8
	italic	9
	bold	10
	bold italic	11
Sans-serif	plain	0
	italic	1
	bold	2
	bold italic	3

Table 8-1 Microtype Fonts (continued)

Family Name	Style	Index Entry For the Family
monospaced	plain	4
	italic	5
	bold	6
	bold italic	7
dialog	plain	0
	italic	1
	bold	2
	bold italic	3
dialog input	plain	0
	italic	1
	bold	2
	bold italic	3

Note

The fonts listed in the preceding table were supplied to Microware by Agfa (a division of Bayer Corporation). They are only for developmental use by our customers and are not for redistribution. Contact Agfa to obtain font sets for your distribution.

Modifying font.properties

PersonalJava Solution for OS-9 uses the file `font.properties` to map logical Java font names (Serif, Dialog...) to the native font names on a particular system. The file `font.properties` is found on your Windows development host at `MWOS\SRC\PJAVA\LIB`

Most of the syntax used in `font.properties` is specified by Sun. There is, however, a portion that can be modified for a particular port of Java. The purpose of this section is to explain the modifications made for the PersonalJava port to OS-9.

Mapping Fonts

Only the mapping between logical and native fonts has been extended.

Syntax

The extended syntax parsed by Java is as follows:

```
<family name>[.<style>].<index>=<font module name>[+<font plug-in
name>][:<font index>],<font type>
```

Parameters

`<family name>`	is the logical font name that can be used by an applet or application
`[.<style>]`	is the font style (optional)
`.<index>`	is the index of this entry for this `<family name>`
=	separates the logical description from the native description
`<font module name>`	is the name of the OS-9 module containing the font. The module name and the module file name must be identical

`[+<font plug-in name>]`	allows the specification of a MicroType plug-in font to be used in conjunction with `<font module name>` (optional)
`[:<font index>]`	allows the specification of a MicroType font index indicating the index of the font inside of `<font module name>` to use If not specified, the index is assumed to be zero (optional)
`,<font type>`	indicates the type of font stored in `<font module name>` The current valid values for `<font type>` include the following: `FONTTYPE_MT` (MicroType) `FONTTYPE_TT` (TrueType)

Example 1

```
sansserif.plain.0=mw_java.fco+mwp_java.fco:0,FONTTYPE_MT
sansserif.italic.0=mw_java.fco+mwp_java.fco:1,FONTTYPE_MT
sansserif.bold.0=mw_java.fco+mwp_java.fco:2,FONTTYPE_MT
sansserif.bolditalic.0=mw_java.fco+mwp_java.fco:3,FONTTYPE_MT
```

In the above example, the font is Microtype sans-serif. Microtype fonts are a compressed format in which several fonts are stored in the same module. Therefore, when specifying them, you need to name the index of the font within the module you wish to use. In this example, index 2 is used for sans-serif bold.

The example also specifies a Microtype Plug-in font (`mwp_java.fco`) containing glyphs that do not change between fonts or font styles. A separate entry is not used for the plug-in because it is handled transparently from Java's perspective.

Example 2

```
serif.plain.0=Times.ttf,FONTTYPE_TT
serif.italic.0=TimesI.ttf,FONTTYPE_TT
serif.bold.0=TimesBD.ttf,FONTTYPE_TT
serif.bolditalic.0=TimesBI.ttf,FONTTYPE_TT
serif.1=Dingbats.ttf,FONTTYPE_TT
```

It is also possible to use True Type fonts. In this example for TrueType fonts, each of the available serif font styles are mapped to a specific native font module. This example also specifies that for any style, `Dingbats.ttf` is used for any character above the glyph range of the Times fonts.

Creating Font Data Modules from Font Files

This section describes the procedure for converting a file to a loadable module for OS-9. This procedure is important in relation to font support for OS-9. The font support files that would normally reside on disk are converted into modules and used directly from memory. To use the files as modules they must be converted from files to modules.

An important concept to understand before performing this conversion is the difference between a file name and a module name. A file name is the name of a file as recorded in the host operating system's directory structure. A module name is the name by which OS-9 will recognize the entity after it has been loaded from disk. Generally, the module name and file name are identical. This, of course, makes it easier to keep track of your OS-9 modules. The file name and module name can differ. For example, you could have a file called `kernel.new` that contains the module called `kernel`. The Microware utility `ident` is used to determine the module contents of a file. For example, an `ident` of `kernel.new` might show:

```
$ ident kernel.new
Header for:       kernel
Module size:      $F310       #62224
Owner:            0.0
Module CRC:       $AD8BC8     Good CRC
Header parity:    $C028       Good parity
Edition:          $55         #85
Ty/La At/Rev      $C01        $A000
Permission:       $555        -----e-r-e-r-e-r
```

```
Exec off:         $A8          #168
Data size:        $1960        #6496
Stack size:       $C00         #3072
Init. data off:   $D5E0        #54752
Data ref. off:    $EF48        #61256
80386 System Mod, Object Code, Sharable, System State
Process
```

Note the module named `kernel` located in a file called `kernel.new`.

This concept of differing file name vs. module name will be employed to create modules for font files. The `mkdatmod` utility is used to wrap the OS-9 module structure around a file. Refer to the ***Utilities Reference*** manual for more information about `mkdatmod`.

Assume you had a file, called `testfile`, that you wanted converted into a module with the same name. The `mkdatmod` command line would be:

```
mkdatmod testfile testfile.mod -tp=arm
```

This creates an ARM module called `testfile` in a file called `testfile.mod`. Running `ident` on `testfile.mod` shows:

```
Header for:       testfile
Module size:      $F390        #62352
Owner:            0.0
Module CRC:       $F720AF      Good CRC
Header parity:    $C3A1        Good parity
Edition:          $1           #1
Ty/La At/Rev      $400         $8000
Permission:       $111         -------r---r---r
Exec off:         $78          #120
ARM Data Mod, Sharable
```

Note the module named `testfile` located in a file called `testfile.mod`.

Localizing Your PersonalJava Solution for OS-9

The PersonalJava Solution package you have installed is localized for the U.S. and Europe. It can be modified after installation for the following languages:

- Japanese
- Korean
- Traditional Chinese
- Chinese
- Thai
- Russian
- Hebrew
- Arabic

To localize the package for one of these locales, the `font.properties` file must be modified to match the version required for the language. Complete the following steps:

Step 1. Go to the directory `E:\MWOS\DOS\jdk1.1.8\lib` on your Windows host. Locate the following files from the Windows hosted JDK:

Table 8-2 Localizing Files

Region	**File**
English	`font.properties`
Japanese	`font.properties.ja`
Korean	`font.properties.ko`

Table 8-2 Localizing Files

Region	File
Traditional Chinese	`font.properties.zh_TW`
Chinese	`font.properties.zh`
Thai	`font.properties.th`
Russian	`font.properties.ru`
Hebrew	`font.properties.iw`
Arabic	`font.properties.ar`

Step 2. Use a text file difference tool to determine the differences between your localization choice and `font.properties`:

```
cd \E:\MWOS\DOS\jdk1.1.8\lib
diff font.properties font.properties.ja
```

Step 3. Copy Microware's original `font.properties` for safe keeping. Consider the following example:

```
cd E:\MWOS\SRC\PJAVA\LIB
copy font.properties font.properties.en
```

Step 4. Edit `font.properties` and make similar changes reported in step 2 by the difference tool. Presumably, you have already purchased the correct font modules from AGFA.

Step 5. Make sure you are using a disk-based system that loads the JVM and its resources from a MWOS directory on a system disk.

- If using a disk-based configuration, FTP the new `font.properties` file to the OS-9 target machine and place it in `/h0/MWOS/SRC/PJAVA/LIB`.

- If using a diskless configuration, The new `font.properties` for the locale you have selected is added to `pjava_home.mar` the next time `pjruntime` is generated.

For More Information

For a complete discussion of `pjruntime`, refer to **Chapter 4: Choosing a PersonalJava Diskless Strategy**.

Modifying awt.properties

PersonalJava Solution for OS-9 uses the file `awt.properties` to control the way the font is used on your device. It is also used to control whether multiple windows are allowed. The file `awt.properties` is found on your Windows development host at `E:\MWOS\SRC\PJAVA\LIB`

Setting colorMode

The `colorMode` property in the `awt.properties` file gives the AWT (Abstract Windowing Toolkit) more information about the way color is implemented on a device.

Syntax

```
AWT.colorMode={color|gray|mono}
```

Options

`color`	the platform has a color display This is the default value if the property is not present.
`gray`	the platform has multiple gray shade that simulates a color display
`mono`	the platform has only two colors

Setting AGFA Font Engine Memory Consumption

The AGFA Font Engine is used by Microware's AWT to render Java text. It has a cache for remembering previous renders and also a buffer space for rendering a specific character. The size of these areas can be customized.

The reason you may change these values is that there always exists a speed and memory trade off. The larger the cache and buffer size the more renderings the engine can store, but inversely, you can save roughly 300K of RAM for the cache size and 20K for the buffer size if you use the minimum sizes.

In the `awt.properties` file, two lines can be added to control the size of the cache and buffer that the AGFA Font Engine uses to render text. Suggested values for highest performance include the following:

```
AWT.agfacachesize=320000
AWT.agfabuffersize=60000
```

To set the values to their lowest recommended size of 25000 and 40000 respectively, change the lines to the following. These are the defaults:

```
AWT.agfacachesize=0
AWT.agfabuffersize=0
```

Using Multiple Windows

PersonalJava Solution for OS-9 provides the ability to use multiple windows and related features.

In the file `awt.properties`, the property `AWT.multiwindow` controls whether or not multiple windows are allowed.

```
# Allow multiple windows
AWT.multiwindow=yes
```

The legal values for `AWT.multiwindow` are `yes` and no. Default is `yes`.

If `AWT.multiwindow` is set to no, the classes `java.awt.CheckboxMenuItem`, `java.awt.Dialog` (modeless), `java.awt.Frame`, `java.awt.Menu`, `java.awt.MenuBar`, `java.awt.MenuShortcut`, and `java.awt.Window` throw an `UnsupportedOperationException` as defined in the PersonalJava specification.

If the value for `AWT.multiwindow` is set to `yes`, `java.awt.CheckboxMenuItem`, `java.awt.Dialog` (modeless and modal), `java.awt.Frame`, `java.awt.Menu`, `java.awt.MenuBar`, `java.awt.MenuShortcut`, and `java.awt.Window` all work as they do for the JDK.

Using Scrollbars

PersonalJava Solution for OS-9 provides the ability to use the optional `Scrollbar` class from AWT.

In the file `awt.properties`, the property `AWT.scrollbar` controls whether or not Scrollbar is allowed.

```
# Allow/disallow Scrollbar class
AWT.scrollbar=yes
```

The legal values for `AWT.scrollbar` are `yes` and no. Default is `yes`.

If `AWT.scrollbar` is set to no, any attempt to instantiate a `Scrollbar` object results in the throwing of `UnsupportedOperationException` as allowed by the PersonalJava specification.

Chapter 9: Monitoring PersonalJava Applications

PersonalJava Solution for OS-9 offers a number of ways to monitor the activities and resource usage of your PersonalJava applications. Details on these various monitoring technologies follow in this chapter, including the following:

- **Memory Usage Monitoring**
- **Native Stack Usage Monitoring**
- **AWT Activities Monitoring**

Memory Usage Monitoring

PersonalJava Solution for OS-9 gives you two ways to instrument the memory usage of your PersonalJava applications:

- `MEMWATCH` environment variable
  The debug version of the JVM (Java Virtual Machine), `pjava_g`, and the debug version of the window manager, `winmgrg`, both print memory usage statistics when they exit if the `MEMWATCH` environment variable is set prior to their start.
- Memory Stopwatch class
  Microware ships a class that allows RAM usage of PersonalJava applications to be monitored. The remainder of this section contains details about this class and an example application to monitor the amount of RAM particular GUI elements consume.

A memory stopwatch gives you the ability to monitor memory activity over a period of time. It works much like a conventional stopwatch as it starts before the activity being watched starts and it is stopped after the activity finishes. The following sections describes the memory stopwatch included in your PersonalJava Solution for OS-9 package.

- **Stopwatch Java API** lists all the Java methods included in the `MemStopWatch` class.
- **The MemStopWatch Example Java Program** provides sample code of a Java program using `MemStopWatch`.
- **Using the MemStopWatch Example Java Program** is a tutorial for using the MemStopWatch example.

Introduction

The `MemStopWatch` class is implemented largely with native methods that monitor activity on the C heap. The C heap is used by the JVM to allocate class-related data structures. It is also heavily used by native methods. For example, the AWT (Abstract Windowing Toolkit) native methods use a combination of C heap (such as `malloc()`, `calloc()`, and `free()`) calls and `_os_srqmem()`.

MemStopWatch monitors the changes in the following items:

- RAM allocated from the C heap via malloc, calloc, and realloc and RAM allocated with _os_srqmem
- Total count of the number of bytes allocated by the allocation functions
- RAM allocated to the processes by OS-9
- Number of segments (separate calls to allocation functions) outstanding
- Number of calls to the various allocation and deallocation functions
- RAM allocated from the Java heap

The stopwatch also keeps track of a number of maximums that are tallied regardless of whether or not the stopwatch is running. These maximums are listed below:

- Maximum RAM allocated from the C heap and via _os_srqmem
- Maximum RAM allocated to the process by OS-9
- Maximum request made of the C heap or _os_srqmem
- Maximum number of segments outstanding at any one time

Stopwatch Java API

The MemStopWatch class is included in the package com.microware.support. It is a subclass of java.lang.Object and it has no class data members. The Stopwatch Java API methods are described on the following pages.

Table 9-1 Constructor

Function	Description
MemStopWatch()	Construct a new MemStopWatch instance

Table 9-2 Stopwatch Methods

Function	Description
`isRunning()`	Check if the stopwatch is running
`start()`	Start monitoring memory activity
`stop()`	Stop monitoring memory activity

Table 9-3 Stopwatch Information Methods

Function	Description
`clear()`	Reset information to zero
`getAddReallocs()`	Return number of enlarging `realloc()` calls
`getAllocTotal()`	Return total number of bytes allocated
`getCallocs()`	Return number of calls to `calloc()`
`getCurrJavaRAM()`	Return current Java heap RAM usage
`getCurrJavaRAM()`	Return current RAM usage
`getCurrSegs()`	Return number of segments currently in use
`getCurrSysRAM()`	Return current system RAM usage
`getFrees()`	Return number of `free()` calls

Table 9-3 Stopwatch Information Methods (continued)

Function	Description
`getMallocs()`	Return number of calls to `malloc()`
`getMaxAlloc()`	Return maximum single allocation
`getMaxRAM()`	Return maximum RAM usage
`getMaxSegs()`	Return maximum number of segments in use
`getMaxSysRAM()`	Return maximum system RAM usage
`getSrqmems()`	Return number of `_os_srqmem()` calls
`getSrtmems()`	Return number of `_os_srtmem()` calls
`getSubReallocs()`	Return number of shrinking `realloc()` calls

Stopwatch information methods retrieve the various pieces of information from the stopwatch object. If the stopwatch is running, these methods return information from the last time the stopwatch was stopped.

Three types of fields are returned when calling these stopwatch information methods:

Delta fields	reflect changes in an item occurring while the stopwatch is running They can have negative values. For example, if more memory is freed while the stopwatch is running than was allocated, `getCurrRAM` would have a negative value.
Counter fields	reflect changes in an item occurring while the stopwatch is running

	Unlike delta fields, they cannot have negative values as they are counters of the number of items that have occurred.
Maximum fields	show the highest value that an item reaches regardless of whether or not the stopwatch is running at the time the value is reached Maximums are always positive.

Table 9-4 Debugging Methods

Function	Description
`toString()`	Generate a string representation of the MemStopWatch
`toString(String title)`	Generate a string representation of the MemStopWatch with an optional title

clear()

Reset Information to Zero

Syntax

```
void clear()
```

Description

`clear()` clears the stopwatch object to zero. If the stopwatch is running, it is stopped.

Exceptions

None

See Also

`start()`

`stop()`

getAddReallocs()

Return Number of Enlarging realloc() Calls

Syntax

```
public int getAddReallocs();
```

Description

`getAddReallocs` is a counter field that returns the number of calls to `realloc` or `_lrealloc` where the new amount of memory was more than was already allocated.

Exceptions

None

See Also

`clear()`

`getSubReallocs()`

`start()`

`stop()`

getAllocTotal()

Return Total Number of Bytes Allocated

Syntax

```
public int getAllocTotal();
```

Description

`getAllocTotal` is a counter field that returns the total count of the number of bytes allocated from the C heap or via `_os_srqmem`.

Exceptions

None

See Also

`clear()`

`getMaxAlloc()`

`start()`

`stop()`

getCallocs()

Return Number of Calls to calloc()

Syntax

```
public int getCallocs();
```

Description

`getCallocs` is a counter field that returns the number of calls to `calloc` or `_lcalloc`.

Exceptions

None

See Also

`clear()`

`getFrees()`

`getMallocs()`

`start()`

`stop()`

getCurrJavaRAM()

Return Current Java Heap RAM Usage

Syntax

```
public int getCurrJavaRAM();
```

Description

`getCurrJavaRAM` is a delta field that returns the amount of RAM currently outstanding from the Java heap.

Exceptions

None

See Also

`clear()`

`start()`

`stop()`

Syntax

```
public int getCurrRAM();
```

Description

getCurrRAM is a delta field that returns the amount of RAM currently outstanding from the C heap and _os_srqmem calls.

Exceptions

None

See Also

clear()

getCurrSysRAM()

getMaxRAM()

start()

stop()

getCurrSegs()

Return Number of Segments Currently in Use

Syntax

```
public int getCurrSegs();
```

Description

`getCurrSegs` is a delta field that returns the number of discrete memory allocations currently outstanding.

Exceptions

None

See Also

`clear()`

`getMaxSegs()`

`start()`

`stop()`

getCurrSysRAM()

Return Current System RAM Usage

Syntax

```
public int getCurrSysRAM();
```

Description

`getCurrSysRAM` is a delta field that returns the amount of RAM currently allocated to the process by OS-9.

Exceptions

None

See Also

`clear()`

`getCurrJavaRAM()`

`getMaxSysRAM()`

`start()`

`stop()`

getFrees()

Return Number of free() Calls

Syntax

```
public int getFrees();
```

Description

`getFrees` is a counter field that returns the number of calls to `free` or `_lfree`.

Exceptions

None

See Also

`clear()`

`getCallocs()`

`getMallocs()`

`start()`

`stop()`

getMallocs()

Return Number of Calls to malloc()

Syntax

```
public int getMallocs();
```

Description

`getMallocs` is a counter field that returns the number of calls to `malloc` or `_lmalloc`.

Exceptions

None

See Also

`clear()`

`getCallocs()`

`getFrees()`

`start()`

`stop()`

getMaxAlloc()

Return Maximum Single Allocation

Syntax

```
public int getMaxAlloc();
```

Description

`getMaxAlloc` is a maximum field that returns the largest single request of the C heap or via `_os_srqmem`.

Exceptions

None

See Also

`clear()`

`getAllocTotal()`

`start()`

`stop()`

getMaxRAM()

Return Maximum RAM Usage

Syntax

```
public int getMaxRAM();
```

Description

`getMaxRAM` is a maximum field that returns the maximum amount of RAM outstanding from the C heap and `_os_srqmem` calls.

Exceptions

None

See Also

`clear()`

`getCurrJavaRAM()`

`getMaxSysRAM()`

`start()`

`stop()`

getMaxSegs()

Return Maximum Number of Segments in Use

Syntax

```
public int getMaxSegs();
```

Description

`getMaxSegs` is a maximum field that returns the maximum number of allocations outstanding at one time.

Exceptions

None

See Also

`clear()`

`getCurrSegs()`

`start()`

`stop()`

getMaxSysRAM()

Return Maximum System RAM Usage

Syntax

```
public int getMaxSysRAM();
```

Description

`getMaxSysRAM` is a maximum field that returns the maximum amount of RAM ever allocated to the process by OS-9.

Exceptions

None

See Also

`clear()`

`getCurrJavaRAM()`

`getCurrSysRAM()`

`start()`

`stop()`

getSrqmems()

Return Number of _os_srqmem() Calls

Syntax

```
public int getSrqmems();
```

Description

`getSrqmems` is a counter field that returns the number of calls to `_os_srqmem`.

Exceptions

None

See Also

`clear()`

`getSrtmems()`

`start()`

`stop()`

Syntax

```
public int getSrtmems();
```

Description

`getSrtmems` is a counter field that returns the number of calls to `_os_srtmem`.

Exceptions

None

See Also

`clear()`

`getSrqmems()`

`start()`

`stop()`

getSubReallocs()

Return Number of Shrinking realloc() Calls

Syntax

```
public int getSubReallocs();
```

Description

`getSubReallocs` is a counter field that returns the number of calls to `realloc` or `_lrealloc` where the new amount of memory was less than was already allocated.

Exceptions

None

See Also

`clear()`

`getAddReallocs()`

`start()`

`stop()`

isRunning()

Check If the Stopwatch is Running

Syntax

```
boolean isRunning()
```

Description

`isRunning()` returns whether or not the stopwatch object is currently running.

Exceptions

None

See Also

`start()`

`stop()`

MemStopWatch()

Construct a New MemStopWatch Instance

Syntax

```
public MemStopWatch()
```

Description

`MemStopWatch()` is the constructor for the object.

Exceptions

`java.lang.OutOfMemory`	insufficient memory to allocate the stopwatch.

Syntax

```
public void start()
```

Description

`start()` starts the monitoring of memory activity for this stopwatch. If the stopwatch is already started, it continues to run.

Exceptions

None

See Also

`stop()`

stop()

Stop Monitoring Memory Activity

Syntax

```
public void stop()
```

Description

`stop()` stops monitoring memory activity for this stopwatch. If the stopwatch is already stopped, it remains stopped.

Exceptions

None

See Also

`start()`

toString()

Generate a string representation of the MemStopWatch

Syntax

```
public String toString();
```

Description

`toString()` converts the stopwatch object into an ASCII representation. If the stopwatch is running, the statistics from the last time it was stopped are used.

An example string might be:

```
Stopped MemStopWatch:
   CurrRAM = 1364382 MaxRAM = 8908704
   CurrSysRAM = 1392640 MaxSysRAM = 9527296
   CurrSegs = 1290 MaxSegs = 4936
   MaxAlloc = 6291464 AllocTotal = 2386336
   Mallocs = 1474 Callocs = 7903 AddReallocs = 1 Srqmems = 13
   Frees = 8099 SubReallocs = 0 Srtmems = 2
   CurrJavaRAM = 28478
```

Exceptions

None

See Also

`toString(String title)`

toString(String title)

Generate a string representation of the MemStopWatch With an Optional Title

Syntax

```
public String toString(String title);
```

Description

`toString(String title)` converts the stopwatch object into an ASCII representation with a title. If the stopwatch is running, the statistics from the last time it was stopped are used.

For example, if title was `After instantiation`, then the string might be:

```
After instantiation:
   CurrRAM = 1364382 MaxRAM = 8908704
   CurrSysRAM = 1392640 MaxSysRAM = 9527296
   CurrSegs = 1290 MaxSegs = 4936
   MaxAlloc = 6291464 AllocTotal = 2386336
   Mallocs = 1474 Callocs = 7903 AddReallocs = 1 Srqmems = 13
   Frees = 8099 SubReallocs = 0 Srtmems = 2
   CurrJavaRAM = 24878
```

Exceptions

None

See Also

`toString()`

The MemStopWatch Example Java Program

This example Java source uses the `MemStopWatch` class to gather memory statistics during the creation and drawing of a user selected Java AWT component:

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import com.microware.support.*;

// this version of MemStopWatch is the same as the enterprise version except
// anything that creates a frame has been removed and the unsupported
// Scrollbar class has been removed from the options list

class Main extends Frame implements ActionListener
{
    String options[] = {"Button", "Canvas", "Checkbox", "Choice", "Label",
                        "List", "Panel", "ScrollPane", "TextArea", "TextField"};

    String separator = "-------------------------------------------------------";

    // reference to the current component displayed on the screen
    Component currentComponent;

    // main panel where the components are displayed
    Panel displayPanel = new Panel();

    // our stop watch
    MemStopWatch stopWatch;

    // TextArea for displaying statistics
    TextArea ta;

    // choice for choosing which widget to display
    Choice choice;

    // show button
    Button showButton;

    Main()
    {
        super("MemStopWatch Test");
        Panel widgetPanel;
        WindowEventHandler wl;

        // create everything
        try
        {
            widgetPanel = new Panel();
            choice = new Choice();
            showButton = new Button("Show");
            wl = new WindowEventHandler();
```

```
            ta = new TextArea();
            stopWatch = new MemStopWatch();
        } catch (Throwable e) {
            System.out.println(e);
            return;
        }

        // fill out the choice widget
        for (int i = 0; i < options.length; i++)
        {
            choice.add(options[i]);
        }

        // set up the show button and add it to the widget panel
        showButton.addActionListener(this);

        widgetPanel.setLayout(new GridLayout(1, 0));
        widgetPanel.add(choice);
        widgetPanel.add(showButton);
        add("North", widgetPanel);

        displayPanel.setBackground(Color.gray);
        add("Center", displayPanel);

        // set up the text area
        ta.setRows(10);
        ta.setEditable(false);
        add("South", ta);

        addWindowListener(wl);

        setSize(400, 350);
        show();
    }

    public boolean changeComponents(String name)
    {
        if (name == null)
            return false;

        // if we are viewing a component hide it, remove it and set the
        // reference to null
        if (currentComponent != null)
        {
            currentComponent.setVisible(false);
            displayPanel.remove(currentComponent);
            currentComponent = null;
        }

        showButton.setEnabled(false);

        // wait for any other threads to finish running
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {};
```

```
// clear the stopwatch and then start it
stopWatch.clear();
stopWatch.start();

// create the new component under the supervision of the stopwatch
try {
    if (name.equals("Button"))
        currentComponent = new Button("TestButton");
    else if (name.equals("Canvas"))
        currentComponent = new mswCanvas();
    else if (name.equals("Checkbox"))
        currentComponent = new Checkbox("Checkbox Example");
    else if (name.equals("Choice"))
    {
        Choice newChoice = new Choice();
        newChoice.addItem("Item 1");
        newChoice.addItem("Item 2");
        currentComponent = newChoice;
    }
    else if (name.equals("Label"))
        currentComponent = new Label("Label text");
    else if (name.equals("List"))
    {
        List newList  = new List(2);
        newList.addItem("First Item");
        newList.addItem("Second Item");
        newList.addItem("Third Item");
        currentComponent = newList;
    }
    else if (name.equals("Panel"))
    {
        currentComponent = new Panel();
        currentComponent.setBackground(Color.blue);
        currentComponent.setSize(100,100);
    }
    else if (name.equals("ScrollPane"))
    {
        currentComponent = new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
        currentComponent.setSize(100,100);
    }
    else if (name.equals("TextArea"))
        currentComponent = new TextArea("TextArea", 7, 10,
        TextArea.SCROLLBARS_BOTH);
    else if (name.equals("TextField"))
        currentComponent = new TextField("TextField", 10);

    } catch (Throwable e) {
        System.out.println("The following exception occurred: "+e);
        stopWatch.stop();
        currentComponent = null;
        showButton.setEnabled(true);
        return false;
    }
```

```
        // set the current component name so we can recognize it again
        currentComponent.setName(name);

        // do not add a window derivative to the container
        if (!(currentComponent instanceof java.awt.Window))
            displayPanel.add("Center", currentComponent);

        // arrange the panel and show the current component
        displayPanel.doLayout();
        currentComponent.setVisible(true);

        // wait for any other threads to finish running
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {};

        // if the watch is running stop it
        if (stopWatch.isRunning())
        {
            stopWatch.stop();
        }

        showButton.setEnabled(true);
        return true;
    }

    public void actionPerformed(ActionEvent evt)
    {
        // see which button was pressed

        if ("Show".equals(evt.getActionCommand()))
        {
            // create the newly selected component
            if (changeComponents(choice.getSelectedItem()))
        {
            ta.append(separator + "\n");
            ta.append(stopWatch.toString("Memory usage since creation of " +
                currentComponent.getName()));
        }
    }
}

public static void main(String args[])
{
    new Main();
}

public class WindowEventHandler extends WindowAdapter
{
    public void windowClosing(WindowEvent evt)
    {
        System.exit(0);
    }
}
```

```
    // canvas used in the Canvas test
    public class mswCanvas extends Canvas
    {
        mswCanvas()
        {
            super();
            setSize(100, 100);
            setBackground(Color.blue);
        }
        public void paint(Graphics g)
        {
            g.drawRect(0, 0, getSize().width-1, getSize().height-1);
        }
    }
}
```

Using the MemStopWatch Example Java Program

In this example, you can select an AWT component from the choice widget in the upper left corner of the frame. When the user presses the `Show` button, the stopwatch is started and a Java AWT component is created. After the component has been drawn, the stopwatch is stopped and the collected memory statistics are printed in the text area.

The MemStopWatch Example Java Source File

The source for this example has been installed on the Windows development machine in the `E:\MWOS\SRC\PJAVA\EXAMPLES\MEMORY_SW` directory.

Compiling the Source File

In a DOS shell on a Windows 95/NT development machine, compile the source file using the Java compiler.

```
> cd \MWOS\SRC\PJAVA\EXAMPLES\MEMORY_SW
> javac -classpath \MWOS\SRC\PJAVA\LIB\classes.zip Main.java
```

Once the compilation succeeds, the following class files are placed in the same directory as the Java source file:

```
Main$WindowEventHandler.class
Main$mswCanvas.class
Main.class
```

Note

If the compilation fails, make sure you typed in and named the program exactly as shown above. Capitalization is important.

Transferring the Class to the Target OS-9 System

In this example, files are transferred using FTP. You can also use NFS. The steps to transfer using FTP are shown below:

Step 1. Choose `Start->Run` on the Windows desktop.

Step 2. In the Run dialog box, enter `ftp <machine name>` then click `OK`.

Step 3. Log on to the OS-9 machine by typing the user name and password in the FTP (MS-DOS Shell) window. The default user name and password for OS-9 machines is `super` and `user`.

Step 4. Change to the directory containing the application classes on the Windows machine by entering the following in the FTP window:

```
lcd \MWOS\SRC\PJAVA\EXAMPLES\MEMORY_SW
```

Step 5. Change to the demo directory on the OS-9 machine by entering the following in the FTP window:

```
cd /h0/MWOS/SRC/PJAVA
```

Step 6. Create the MEMORY_SW directory on the OS-9 machine by entering the following in the FTP window:

```
mkdir EXAMPLES
mkdir EXAMPLES/MEMORY_SW
```

Step 7. Change to the MEMORY_SW directory on the OS-9 machine by entering the following in the FTP window:

```
cd EXAMPLES/MEMORY_SW
```

Step 8. Change to binary transfer by entering the following in the FTP window:

```
bin
```

Step 9. Transfer the class files by entering the following in the FTP window:

```
mput *.class
```

Answer `yes` to each prompt.

Step 10. Quit the FTP session.

Starting the Java Application on the OS-9 System

Start the Java application by using telnet to communicate with the OS-9 system:

Step 1. Choose `Start->Run` from the Windows desktop.

Step 2. In the Run window text field enter `telnet <target system>` and click `OK`.

Step 3. Log onto the OS-9 system by entering the user name and password. `super` and `user` are the defaults for OS-9 systems.

Step 4. Change to the demo directory on the OS-9 machine by entering

```
cd /h0/MWOS/SRC/PJAVA/EXAMPLES/MEMORY_SW
```

Step 5. Run the application on the OS-9 target system using the debug version of the Java interpreter, `pjava_g`. Only the debug version of the JVM includes the code necessary to monitor memory usage with the MemStopWatch class.

```
pjava_g Main &
```

A window appears containing the following:

- a `Choice` widget in the upper left corner of the frame
- a `Show` button in the upper right corner of the frame
- a Panel in the middle of the frame
- a TextArea at the bottom of the frame

To use the application to retrieve memory statistics, select a widget name from the `Choice` widget and press `Show`.

Native Stack Usage Monitoring

The Microware PersonalJava Virtual Machine incorporates the StackWatch feature to measure the amount of native stack used by threads within a particular application. Using this technique enables developers to minimize the amount of memory used by their Java programs.

This section includes the following parts:

Introduction

StackWatch is a diagnostic utility built into the *debug* version of Microware's PersonalJava VM (`pjava_g`). StackWatch tells you how much memory each VM thread is using in its native (or C) stack.

This information helps optimize memory use in the following two ways:

- If the application is using inordinate amounts of memory in its native stack, it may require tuning to use less memory.
- If the application never uses more than a fraction of the memory allocated to its native stacks, the size of those stacks can be reduced.

Using StackWatch

Use the *debug* version of the PersonalJava VM to use StackWatch. This executable is called `pjava_g` in PersonalJava Solution for OS-9.

To enable StackWatch, define an environment variable called `STACKWATCH` before starting the VM. When the VM starts the Java application it will notice the `STACKWATCH` environment variable and enable the StackWatch feature.

The Java application can then be run and exited normally. When the VM shuts down, the StackWatch feature prints out a summary of the stacks that were created for each thread that existed during the run of the application. Following is a typical scenario:

```
$ setenv STACKWATCH 1
$ pjava_g ThreadTest

   C-Stack usage by threads
   Free     Used     Size   Status   Name
 -------- -------- -------- -------  ------------------------
  126164     4076   130240     dead  "2"
  126540     3700   130240     dead  "1"
  126540     3700   130240     dead  "0"
  129208     1032   130240  running  "Idle thread"
  129756      484   130240  running  "Async Garbage Collector"
  129768      472   130240  running  "Finalizer thread"
  129804      436   130240  running  "Clock"
```

For each thread started during execution of the application, StackWatch prints out the following information:

- Free - amount of stack not used by the thread
- Used - largest amount of stack ever used by the thread
- Size - total amount of memory allocated for the thread’s stack
- Status - current status of the thread
- Name - name assigned by your application or the VM to the thread

Interpreting the Results

From the previous example, it can be seen that no thread used more than 5K (where 1K is 1024 bytes) of native stack. The StackWatch summary in **Figure 9-1** shows the PersonalJava VM started with a stack size of 5K (where 1K is 1024 bytes) for each stack. If this run represents a worst-case scenario for stack usage, the stack size for each thread in the application could be reduced from 128K to 5K, saving roughly 850K of RAM.

Figure 9-1 StackWatch Sample Summary

```
$ setenv STACKWATCH 1
$ pjava_g -ss5k ThreadTest

   C-Stack usage by threads
   Free     Used     Size   Status   Name
 -------- -------- -------- -------  --------------------------
     212     4076     4288     dead  "2"
     588     3700     4288     dead  "1"
     588     3700     4288     dead  "0"
    3256     1032     4288  running  "Idle thread"
    3804      484     4288  running  "Async Garbage Collector"
    3816      472     4288  running  "Finalizer thread"
    3844      444     4288  running  "Clock"
```

AWT Activities Monitoring

The activities of AWT and Microware's AFW class libraries can be monitored by using the appdbg facility.

The application debugging (`appdbg`) environment provides support for applications to convey debugging information to the user. This debugging information is invaluable in determining the order in which events took place.

This chapter includes the following topics:

The appdbg Environment

`appdbg` has the following attributes:

- fast — `appdbg` is very efficient at writing messages. This is important when timing is an issue. For example, if the debugging version of an application runs several times slower than the non-debug version, its behavior in relation to other applications can be different. Writing a message does introduce overhead, but Microware worked to reduce this overhead.
- thread-safe — `appdbg` is written with no static information so it can be called by different threads at the same time. For example, if the main-line code is in the process of printing a message when a signal arrives, the signal handler is free to emit debugging information. In addition, `appdbg` is non-blocking. This is important for user-state cooperative threading.

- multi-module, multi-process

  appdbg supports the emitting of debug information by any number of applications at the same time All the information is gathered in a common place so you can examine the actual order of events in different programs. This can be very helpful in the presence of inter-process communication. Each message can be prepended with the module name and/or process ID of the application emitting the information.

For More Information

Refer to **appdbg Environment Variables** on page 201 for more information about these messages.

- unbuffered — appdbg does not buffer information. This allows for an accurate log in the presence of application failures.
- non-intrusive — You can c hoose appdbg to always be enabled. This enables you to examine the tail end of the debug information written without having to consciously enable debugging at the beginning of the run.

appdbg Files

The following files are used with appdbg:

<os>/<proc>/CMDS/appdbg_trap

system-state trap handler that must be present on the target system

<os>/<proc>/CMDS/adump

utility used to display debug information

Using appdbg

This section describes how to enable the debug information and choose the type of information you want to see. This section assumes you have an application that was compiled with appdbg information. Prior to executing any applications containing appdbg there are several environment variables you can set to control appdbg.

appdbg Environment Variables

APPDBG_MOD <name>[,<size>]

is set to the name of the module to use for debugging information and, optionally, the size of the module

The size determines how much historical information is available at any given time.

<name> is the name of the module to use

<size> is the optional decimal size of the module in K bytes

For example, setenv APPDBG_MOD dbglog would set the module name used to dbglog and setenv APPDBG_MOD runlog,256 would set the module name used to runlog and the size to 256K bytes. The default module name is appdbg_mod and the default module size is 512K. The minimum module size is 16K.

APPDBG_LVL <level>

is used to set the level of debug information you want to view

Generally, the higher the level, the more detailed and verbose information you see. The extra amount of information seen varies from application to application. The format of the string used to set APPDBG_LVL is a decimal number For example, setenv

	`APPDBG_LVL 2` would set the information level to two. If `APPDBG_LVL` is not set or is set to `0`, no debug information is emitted.
`APPDBG_OPS <opts>`	is set to indicate options about how the information is emitted The format is a sequence of characters representing options. The following two options are valid: `m` - prepend each line of each message with the name of the module emitting the message. This is useful when multiple applications are running that emit debug information from the same library. p - prepend each line of each message with the process ID of the application emitting the message This is useful when multiple copies of the same application are running simultaneously. Either option can be used alone or together. For example, `setenv APPDBG_OPS m` causes the module name to be prepended and `setenv APPDBG_OPS mp` causes both the module name and process ID to be prepended.

`XXXX_MASK <value>`	Every separate sub-system or library has a mask environment variable to filter the nature of information emitted
	See the documentation for the object to determine the values of the bits and the name of the environment variable.
	For example, if your `OAHU` and `XPRESO` packages have debug information you might set the environment variables:
	`setenv OAHU_MASK 0x86`
	`setenv XPRESO_MASK 0xc001c0de`

	All the emitted information is written to a memory module. The contents of the memory module are interpreted by the `adump` (`appdbg` dump) utility.
`AFW_MASK`	specifies the message mask for the `appdbg` information emitted by the debugging version of the Application Framework library (linked to `pjava_g`) The following are the valid bits: `0x001` position and dimension information `0x002` event handling/generation `0x004` object construction and destruction `0x008` draw related operations `0x010` message transmission and reception `0x020` focus related functions and events `0x040` signal subsystem and related functions `0x080` mailbox and internal queue subsystems `0x100` font technology specific functions `0x200` window manager related functions `0x400` information setting or getting functions `0x800` clipboard related functions
`MAWT_MASK`	specifies the message mask for the `appdbg` information emitted by debugging version of the Microware AWT peer implementation (`pjava_g`) The following are the valid bits: `0x0001` position and dimension information `0x0002` event handling/generation

`0x0004` object construction and destruction

`0x0008` draw related operations

`0x0010` message transmission and reception

`0x0020` focus related functions and events

`0x1000` trace information, function entry and exit

`0x2000` color operations

`0x4000` image operations

The adump Utility

There are several modes in which the adump utility can run. These are illustrated below.

adump Modes

Default Mode

You can examine the end of all the messages written. By default, adump writes all the information currently available in the module to standard output. This is used, for example, after the application has terminated to see what it did last. The command line to show the information is one of the following:

```
adump
```

or

```
adump -t
```

If you want the information written to a file so it can be viewed with an editor, use:

```
adump >log.file
```

Then display log.file in an editor to scan through the information.

Background Run Mode

You can keep a continuous log of all the messages written. adump can be run in the background, periodically reading the new information from the module and writing it to standard output (generally redirected to a file). Each time the module contains ¼ new information, adump runs and writes the new quarter of the module out. The command line used might include the following:

```
adump -r >log &
```

Then, run the application(s) containing appdbg information.

Note

Due to the OS-9 EOF lock mechanism, `adump` must be killed before the file can be edited. After the applications generating information have terminated, kill `adump` with the shell `kill` command (see the section **adump Miscellaneous Functions** on page 207 for more information). Only then can the log file be examined.

Polling Mode

Show the debug information as it is written. `adump` can be run such that it reads new information from the debug module when it becomes available and writes it to standard output. This allows debug information to be seen at about the same time it is written (`adump` polls the module 25 times per second). Generally, this mode is used with `adump` running in the foreground. The application is started, then `adump` is run in the polling mode:

```
adump -p
```

This gives the impression that the applications are writing messages to standard output. The advantage is that the output can be stopped without stopping the applications.

Any combination of these scenarios can be used together. For example, you can look at the tail first to decide if you want a continuous log from that point on.

adump Miscellaneous Functions

Adump has two other miscellaneous functions it can perform: it can clear the module and flushing the information.

Clearing the Module

The following command line clears the contents of the module:

```
adump -c=reset
```

This clearing is very useful when debugging event-driven applications. The module is cleared at convenient times to avoid having to examine a great deal of old information.

Flushing the Information

The command line:

```
adump -c=flush
```

signals any `adump` running in the background with the `-r` option to write all the information available.

Note

It is important to do this before killing a background `adump`. This ensures the file it is writing to contains all the debug messages written before it was killed.

Chapter 10: Working with Remote Classes

This chapter describes how to use the remote class loading feature of PersonalJava Solution 3.1. It includes the following topics:

What is Remote Class Loading?

PersonalJava Solution 3.1 enables you to load classes from an HTTP server. This allows applications on remote devices to access a large amount of Java code without requiring the code to reside on the device. In addition, remote class loading enables application code to be updated on the server only. This eliminates the necessity of updating the code on each device.

Configuring Remote Class Loading

Use the `remote_classes.properties` file to set up remote class loading. The file is found in the `E:\MWOS/SRC/PJAVA/LIB` directory. This file specifies the base URL where the classes reside and then defines the particular file that contains a particular class. Following is an excerpt of the `remote-classes.properties` file as shipped by Sun.

```
# @(#)remote-classes.properties1.1 98/02/11
#
#  Copyright (c) 1998 by Sun Microsystems Inc
#
# Properties for defining the location of remote classes
#

# codebase is the URL where files containing the
# remote clases are located.
#
codebase=http://wombat/JavaKin/pjavaRemoteClasses/

# One entry is needed for each remote class and specifies the name of the class
# and the name of the file that the class is in. The file is located using
# the codebase URL set above.
#
# Classes should be packaged intelligently so unnecessary classes are not
# pulled in. The packaging below is just an example and is not very effecient.
# You can package the classes into smaller packages. For example, RMI should
# be separated into a server package, a client package, and a core package
# (needed by both servers and clients).
#

# java.rmi.* classes are in javarmi.zip
#
java/rmi/AccessException=javarmi.zip
java/rmi/AlreadyBoundException=javarmi.zip
java/rmi/ConnectException=javarmi.zip
java/rmi/ConnectIOException=javarmi.zip
java/rmi/MarshalException=javarmi.zip
java/rmi/Naming=javarmi.zip
java/rmi/NoSuchObjectException=javarmi.zip
java/rmi/NotBoundException=javarmi.zip
java/rmi/RMISecurityException=javarmi.zip
java/rmi/RMISecurityManager=javarmi.zip
java/rmi/Remote=javarmi.zip
java/rmi/RemoteException=javarmi.zip
java/rmi/ServerError=javarmi.zip
java/rmi/ServerException=javarmi.zip
java/rmi/ServerRuntimeException=javarmi.zip
java/rmi/StubNotFoundException=javarmi.zip
java/rmi/UnexpectedException=javarmi.zip
java/rmi/UnknownHostException=javarmi.zip
java/rmi/UnmarshalException=javarmi.zip
java/rmi/dgc/DGC=javarmi.zip
java/rmi/dgc/Lease=javarmi.zip
java/rmi/dgc/VMID=javarmi.zip
java/rmi/registry/LocateRegistry=javarmi.zip
java/rmi/registry/Registry=javarmi.zip
java/rmi/registry/RegistryHandler=javarmi.zip
java/rmi/server/ExportException=javarmi.zip
java/rmi/server/LoaderHandler=javarmi.zip
```

```
java/rmi/server/LogStream=javarmi.zip
java/rmi/server/ObjID=javarmi.zip
java/rmi/server/Operation=javarmi.zip
java/rmi/server/RMLoader=javarmi.zip
java/rmi/server/RMIFailureHandler=javarmi.zip
java/rmi/server/RMISocketFactory=javarmi.zip
java/rmi/server/RemoteCall=javarmi.zip
java/rmi/server/RemoteObject=javarmi.zip
java/rmi/server/RemoteRef=javarmi.zip
java/rmi/server/RemoteServer=javarmi.zip
java/rmi/server/RemoteStub=javarmi.zip
java/rmi/server/ServerCloneException=javarmi.zip
java/rmi/server/ServerNotActiveException=javarmi.zip
java/rmi/server/ServerRef=javarmi.zip
java/rmi/server/Skeleton=javarmi.zip
java/rmi/server/SkeletonMismatchException=javarmi.zip
java/rmi/server/SkeletonNotFoundException=javarmi.zip
java/rmi/server/SocketSecurityException=javarmi.zip
java/rmi/server/UID=javarmi.zip
java/rmi/server/UnicastRemoteObject=javarmi.zip
java/rmi/server/Unreferenced=javarmi.zip

# sun.rmi.* classes are in sunrmi.zip
#
sun/rmi/registry/RegistryHandler=sunrmi.zip
sun/rmi/registry/RegistryImpl=sunrmi.zip
sun/rmi/registry/RegistryImpl_Stub=sunrmi.zip
sun/rmi/registry/RegistryImpl_Skel=sunrmi.zip
sun/rmi/server/Dispatcher=sunrmi.zip
sun/rmi/server/LoaderHandler=sunrmi.zip
sun/rmi/server/MarshalInputStream=sunrmi.zip
sun/rmi/server/MarshalOutputStream=sunrmi.zip
sun/rmi/server/RemoteProxy=sunrmi.zip
sun/rmi/server/RMLoader=sunrmi.zip
sun/rmi/server/UnicastRef=sunrmi.zip
sun/rmi/server/UnicastServerRef=sunrmi.zip
sun/rmi/transport/Channel=sunrmi.zip
sun/rmi/transport/Connection=sunrmi.zip
sun/rmi/transport/ConnectionInputStream=sunrmi.zip
sun/rmi/transport/IncomingRefTableEntry=sunrmi.zip
sun/rmi/transport/ConnectionOutputStream=sunrmi.zip
sun/rmi/transport/DGCAckHandler=sunrmi.zip
sun/rmi/transport/DGCClient=sunrmi.zip
sun/rmi/transport/DGCClient$CountTableEntry=sunrmi.zip
sun/rmi/transport/DGCClient$CleanRequest=sunrmi.zip
sun/rmi/transport/DGCClient$LeaseTableEntry=sunrmi.zip
sun/rmi/transport/DGCClient$LeaseRenewer=sunrmi.zip
sun/rmi/transport/DGCImpl=sunrmi.zip
sun/rmi/transport/DGCImpl$LeaseChecker=sunrmi.zip
sun/rmi/transport/DGCImpl$LeaseInfo=sunrmi.zip
sun/rmi/transport/Utils=sunrmi.zip
sun/rmi/transport/Endpoint=sunrmi.zip
sun/rmi/transport/LiveRef=sunrmi.zip
sun/rmi/transport/LocateDGC=sunrmi.zip
sun/rmi/transport/Notifiable=sunrmi.zip
sun/rmi/transport/Notifier=sunrmi.zip
sun/rmi/transport/ObjectTable=sunrmi.zip
sun/rmi/transport/KeepAlive=sunrmi.zip
sun/rmi/transport/Reaper=sunrmi.zip
sun/rmi/transport/RMIThread=sunrmi.zip
sun/rmi/transport/StreamRemoteCall=sunrmi.zip
sun/rmi/transport/Target=sunrmi.zip
sun/rmi/transport/SequenceEntry=sunrmi.zip
sun/rmi/transport/UnreferencedObj=sunrmi.zip
sun/rmi/transport/Transport=sunrmi.zip
sun/rmi/transport/TransportConstants=sunrmi.zip
sun/rmi/transport/WeakRef=sunrmi.zip
```

```
sun/rmi/transport/proxy/CGIClientException=sunrmi.zip
sun/rmi/transport/proxy/CGIServerException=sunrmi.zip
sun/rmi/transport/proxy/CGICommandHandler=sunrmi.zip
sun/rmi/transport/proxy/CGIHandler=sunrmi.zip
sun/rmi/transport/proxy/CGIForwardCommand=sunrmi.zip
sun/rmi/transport/proxy/CGIGethostnameCommand=sunrmi.zip
sun/rmi/transport/proxy/CGIPingCommand=sunrmi.zip
sun/rmi/transport/proxy/CGITryHostnameCommand=sunrmi.zip
sun/rmi/transport/proxy/HttpAwareServerSocket=sunrmi.zip
sun/rmi/transport/proxy/HttpInputStream=sunrmi.zip
sun/rmi/transport/proxy/HttpOutputStream=sunrmi.zip
sun/rmi/transport/proxy/HttpReceiveSocket=sunrmi.zip
sun/rmi/transport/proxy/HttpSendInputStream=sunrmi.zip
sun/rmi/transport/proxy/HttpSendOutputStream=sunrmi.zip
sun/rmi/transport/proxy/HttpSendSocket=sunrmi.zip
sun/rmi/transport/proxy/RMIDirectSocketFactory=sunrmi.zip
sun/rmi/transport/proxy/RMIHttpToCGISocketFactory=sunrmi.zip
sun/rmi/transport/proxy/RMIHttpToPortSocketFactory=sunrmi.zip
sun/rmi/transport/proxy/RMIMasterSocketFactory=sunrmi.zip
sun/rmi/transport/proxy/AsyncConnector=sunrmi.zip
sun/rmi/transport/proxy/RMISocketInfo=sunrmi.zip
sun/rmi/transport/proxy/WrappedSocket=sunrmi.zip
sun/rmi/transport/tcp/ConnectionMultiplexer=sunrmi.zip
sun/rmi/transport/tcp/MultiplexConnectionInfo=sunrmi.zip
sun/rmi/transport/tcp/MultiplexInputStream=sunrmi.zip
sun/rmi/transport/tcp/MultiplexOutputStream=sunrmi.zip
sun/rmi/transport/tcp/InEntry=sunrmi.zip
sun/rmi/transport/tcp/TCPChannel=sunrmi.zip
sun/rmi/transport/tcp/ConnectionAcceptor=sunrmi.zip
sun/rmi/transport/tcp/TCPConnection=sunrmi.zip
sun/rmi/transport/tcp/TCPEndpoint=sunrmi.zip
sun/rmi/transport/tcp/Pinger=sunrmi.zip
sun/rmi/transport/tcp/TCPTransport=sunrmi.zip
sun/rmi/transport/DGCImpl_Stub=sunrmi.zip
sun/rmi/transport/DGCImpl_Skel=sunrmi.zip

# java.sql.* classes are in javasql.zip
#
java/sql/CallableStatement=javasql.zip
java/sql/Connection=javasql.zip
java/sql/DataTruncation=javasql.zip
java/sql/DatabaseMetaData=javasql.zip
java/sql/Date=javasql.zip
java/sql/Driver=javasql.zip
java/sql/DriverInfo=javasql.zip
java/sql/DriverManager=javasql.zip
java/sql/DriverPropertyInfo=javasql.zip
java/sql/PreparedStatement=javasql.zip
java/sql/ResultSet=javasql.zip
java/sql/ResultSetMetaData=javasql.zip
java/sql/SQLException=javasql.zip
java/sql/SQLWarning=javasql.zip
java/sql/Statement=javasql.zip
java/sql/Time=javasql.zip
java/sql/Timestamp=javasql.zip
java/sql/Types=javasql.zip

# java.math.* classes are in javamath.zip
#
java/math/BigDecimal=javamath.zip
java/math/BigInteger=javamath.zip

# sun.tools.debug.* and sun.tools.java.* classes are in debugagent.zip
#
sun/tools/debug/MainThread=debugagent.zip
sun/tools/debug/ThreadList=debugagent.zip
sun/tools/debug/Agent=debugagent.zip
```

```
sun/tools/debug/CommandThread=debugagent.zip
sun/tools/debug/AgentConstants=debugagent.zip
sun/tools/debug/BreakpointHandler=debugagent.zip
sun/tools/debug/BreakpointQueue=debugagent.zip
sun/tools/debug/Field=debugagent.zip
sun/tools/debug/LineNumber=debugagent.zip
sun/tools/debug/LocalVariable=debugagent.zip
sun/tools/debug/StackFrame=debugagent.zip
sun/tools/debug/StepHandler=debugagent.zip
sun/tools/debug/StepConstants=debugagent.zip
sun/tools/debug/StepRequest=debugagent.zip
sun/tools/debug/AgentOutputStream=debugagent.zip
sun/tools/debug/ResponseStream=debugagent.zip
sun/tools/debug/BreakpointSet=debugagent.zip
sun/tools/java/RuntimeConstants=debugagent.zip
sun/tools/java/Constants=debugagent.zip
sun/tools/java/Package=debugagent.zip
sun/tools/java/ClassPath=debugagent.zip
sun/tools/java/ClassPathEntry=debugagent.zip
sun/tools/java/ClassFile=debugagent.zip
sun/tools/java/Identifier=debugagent.zip
sun/tools/java/Type=debugagent.zip
sun/tools/java/ArrayType=debugagent.zip
sun/tools/java/CompilerError=debugagent.zip
sun/tools/java/ClassType=debugagent.zip
sun/tools/java/MethodType=debugagent.zip
```

The URL used as the base location for loading remote classes is specified by the `codebase` key word. As described in the property file comments, classes can be broken into Zip files at the user's discretion. The next section describes how to construct different Zip files.

Building Remote Class Zip Files

Class zip files are distributed with the PersonalJava Solution package in the `MWOS/SRC/PJAVA/LIB` directory. These zip files include:

- `classes.zip` contains classes for the standard Java packages
- `javamath.zip` contains classes for the `java.math` package
- `javarmi.zip` contains classes for the `java.rmi` package
- `javasql.zip` contains classes for the `java.sql` package
- `sunrmi.zip` contains classes for the `sun.rmi` package

To reorganize classes into different Zip files, unzip the provided Zip files into a temporary directory. Then recombine classes as desired by putting them into new zip files.

Appendix A: Running PersonalJava Applets

This appendix includes the following topics:

Overview

Microware's PersonalJava Solution for OS-9 includes support for executing PersonalJava applets in a MAUI window, under a MAUI application. This feature is useful for native browser applications, though it is not limited to them. In addition, general purpose legacy MAUI windowing applications can be extended to include windows that contain PersonalJava applets.

Support for executing PersonalJava applets comes in the form of a header file (`\mwos\SRC\DEFS\LIB\mwas.h`) and a library (`\mwos\OS9000\<proc>\LIB\mwas.l`). The header file contains all of the necessary structure definitions and function prototypes. The library resolves the external functions prototyped in the header file.

Example Code

Once you have created a MAUI window ID for the window in which you would like an applet displayed, you may use the example code below to start and destroy an applet. (Error checking code has been eliminated in this example to improve readability.)

```
char *args[] = {
    "code", "CaffeineMarkApplet",
    "width", "200",
    "height", "200",
    NULL
};
mwas_applet_t applet_handle;
void start_applet(WIN_ID window)
{
    mwas_init(NULL, NULL);
    mwas_applet_load(&applet_handle, window,
```

```
"http://www.pendragon-software.com/pendragon/cm3/runtes
t.html", args);
    mwas_applet_init(&applet_handle);
    mwas_applet_start(&applet_handle);
}
void destroy_applet(void)
{
    mwas_applet_stop(&applet_handle);
    mwas_applet_destroy(&applet_handle);
    mwas_applet_dispose(&applet_handle);
}
```

Data Structure

The following data structure is located in the header file `mwas.h`, found in the following location:

`\mwos\SRC\DEFS\LIB\mwas.h`

mwas_applet_t — Maintain applet state

```
typedef struct mwas_applet_t {
    u_int32 sync;       /* sync code for handle */
    u_int32 flags;      /* various flags */
    u_int32 appletid;   /* applet ID returned from load
                           request */
} mwas_applet_t;
```

Description

`mwas_applet_t` is used internally by `mwas.l` to maintain the state of the applet. The application must not modify the contents of this structure. A structure of this format is initialized by the `mwas_applet_load()` call; the storage for the initialized structure must stay intact until `mwas_applet_dispose()` is called.

Functions

The following function calls are located in the `mwas.h` header file, found in the following location:

```
\mwos\SRC\DEFS\LIB\mwas.h
```

`mwas_init()` — Initialize applet support

Syntax

```
mwas_init (char **argvs, char **envps)
```

Description

`mwas_init()` initializes applet support for the calling application.

Note

This call must be made before any other applet support calls can be executed.

`argvs`	a NULL terminated array of pointers to strings These command line arguments are added to the command line for the executed pjava process. `argvs` may be NULL if no additional command line arguments are provided.
`envps`	a NULL terminated array of pointers to strings The strings should be in the format, `<name>=<value>`, where `<name>` is the name of the environment variable and `<value>` is the value for the environment variable. `envps` may be NULL if no additional environment variables are provided.

Returns

If the call is successful, SUCCESS is returned.

If the call is not successful, an error number is returned.

Possible Errors

- EALREADY: mwas_init was called without an intervening call to mwas_term
- indirect errors from malloc, socket, bind, listen, accept, or _os_exec

mwas_applet_load() — Create an instance of an applet

Syntax

```
mwas_applet_load(mwas_applet_t *handle,
WIN_ID parent, char *URL, char **attributes);
```

Description

`mwas_applet_load()` is used to create an instance of a PersonalJava applet. Upon successful return, the handle is initialized and the applet moves into the "loaded" state, though the user's applet code is not yet called. The applet will appear in the windows specified by `parent` as a gray rectangle.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by `handle` should never be modified by the calling application.
`parent`	a MAUI window ID for the window in which the applet should be rendered `parent` must be the correct size for the applet (as specified by the "width" and "height" attributes of the <applet> tag).
`URL`	the fully qualified URL for the page that contained the applet
`attributes`	a NULL terminated array of pointers to strings The strings are the attributes as they appear in the `<applet>` and `<param>` tags. The strings are interpreted as pairs, the even numbered strings are names of attributes, and the odd numbered strings are values for the corresponding name. An even number of strings must appear in the array. `<param>` tags are expressed with "param"

as the even numbered string and "<name>=<value>" are expressed as the odd numbered string. The required attributes include code, height, and width.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: `handle` is NULL, no attributes are specified, an odd number of attributes is specified, or the '=' is missing from the value of a "param" attribute
- `EALREADY`: the handle already appears to refer to an executing applet
- indirect errors from `malloc`, `_os_read`, `_os_write`, or `_os_exec`

mwas_applet_init() Set an applet to the initialized state

Syntax

```
mwas_applet_init(mwas_applet_t *handle);
```

Description

`mwas_applet_init()` is used to set an applet to the initialized state. The applet writer's `init()` method will be called. The applet must either be in the loaded or destroyed state. Upon successful return, the applet moves into the initialized state.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by handle should never be modified by the calling application.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: `handle` is NULL or does not appear to be a valid applet handle
- EALREADY: the applet is not in a valid state for this call
- indirect errors from `malloc` and `_os_write`

mwas_applet_start() — Set an applet to started state

Syntax

```
error_code mwas_applet_start(mwas_applet_t *handle);
```

Description

`mwas_applet_start()` is used to set an applet to the started state. The applet writer's `start()` method will be called. The applet must either be in the initialized or stopped state. Upon successful return, the applet moves into the started state.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by `handle` should never be modified by the calling application.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: handle is NULL or does not appear to be a valid applet handle
- EALREADY: the applet is not in a valid state for this call
- indirect errors from `malloc` and `_os_write`

mwas_applet_stop() — Set an applet to stopped state

Syntax

```
mwas_applet_stop(mwas_applet_t *handle);
```

Description

`mwas_applet_stop()` is used to set an applet to the stopped state. The applet writer's `stop()` method will be called. The applet must be in the started state. Upon successful return, the applet moves into the stopped state.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by handle should never be modified by the calling application.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: `handle` is NULL or does not appear to be a valid applet handle
- `EALREADY`: the applet is not in a valid state for this call
- indirect errors from `malloc` and `_os_write`

mwas_applet_destroy() — Set an applet to destroyed state

Syntax

```
error_code mwas_applet_destroy(mwas_applet_t *handle);
```

Description

`mwas_applet_destroy()` is used to set an applet to the destroyed state. The applet writer's `destroy()` method will be called. The applet must be in the stopped state. Upon successful return, the applet moves into the destroyed state.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by handle should never be modified by the calling application.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: handle is NULL or doesn't appear to be a valid applet handle
- `EALREADY`: the applet is not in a valid state for this call.
- indirect errors from malloc and `_os_write`

mwas_applet_dispose() Dispose of an applet

Syntax

```
mwas_applet_dispose(mwas_applet_t *handle);
```

Description

`mwas_applet_dispose()` is used to dispose of an applet. The applet must be in the destroyed state. Upon successful return, the applet will no longer be displayed and the handle can be reused or freed by the calling application.

`handle`	a pointer to a `mwas_applet_t` data structure The contents of the memory pointed to by handle should never be modified by the calling application.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- `EOS_PARAM`: handle is NULL or doesn't appear to be a valid applet handle
- `EALREADY`: the applet is not in a valid state for this call.
- indirect errors from malloc and `_os_write`

mwas_urlpoll() Check Applets for Request to Visit Web Page

Syntax

```
mwas_urlpoll (char **url, char **target);
```

Description

`mwas_urlpoll()` checks the applets for a request to visit a web page (URL). If any applet has called the `showDocument()` method, the specified URL and optional target is returned to the calling application.

The strings pointed to by `*url` and `*target` should be used before a subsequent call to `mwas)urlpoll()`. Undefined behavior results if valued returned by an old call are used.

This function automatically governs the calls to PJava to ensure that no more than three requests per second are issued. This prevents consuming a lot of system resources needlessly.

`url`	pointer to a pointer to set at the URL string If no URL has been requested by an applet, the pointer at url is set to NULL.
`target`	pointer to a pointer to set to point at the optional target If no target was specified by the applet, the pointer at target is set to NULL.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- EALREADY if applet support is not initialized
- indirect errors from malloc, `_os_read`, `_os_write`, or `_os_getsys`

mwas_term() — Terminates applet support

Syntax

```
mwas_term(void);
```

Description

`mwas_term()` terminates applet support for the calling application. No further calls to applet support functions can be made. `mwas_init()` may be called again to reinitialize support.

Returns

If the call is sucessful, SUCCESS will be returned.

If the call is not successful, an error number will be returned.

Possible Errors

- EALREADY: `mwas_init` has not been called
- indirect errors from `_os_write` or `_os_send`

Appendix B: Running MAUI Applications in Java Windows

RadiSys.
MICROWARE SOFTWARE

Getting the MAUI Window ID

PersonalJava Solution for OS-9 allows you to get the MAUI window ID for a Java component. This allows the writing of MAUI applications that can reparent themselves into the window and exist under control of the Java application.

For example, if you have a legacy windowing MAUI application, such as a map renderer, it can appear in a Java window, under Java's control. In addition, the window can be easily moved or hidden by the Java application.

One way to communicate the window ID to the MAUI process is via a command line parameter, such as `-w`. The Java code necessary to support this facility might look similar to the code below (assuming `panel3` is the location in which you would like the legacy MAUI application to display):

```
// get the window ID of panel3

 MAWTComponentPeer peer =
(MAWTComponentPeer)panel3.getPeer();

int winid = peer.getWinID();

// start the map renderer with additional command
line parameter

Process child = Runtime.getRuntime().exec("drawmap
-w=" +

Integer.toHexString(winid));
```

The Java application and MAUI process can now coexist on the same display device. Furthermore, the Java process can control the MAUI process display window via `panel3`.

Appendix C: Mouse Move Events

An application may need to retain all mouse move events received from the system. This appendix describes why retaining all mouse move events may be necessary. It also discusses how to use the example Java source file `SimpleEventQueue.java` included in your PersonalJava Solution for OS-9 package.

This appendix includes the following topics:

- **Introduction**
- **Contents of this Appendix**
- **The SimpleEventQueue Example Java Program**
- **Using the SimpleEventQueue Example Java classes**

Introduction

The standard behavior of `EventQueue.java` is to compress mouse move events. This means only the last position is posted if multiple mouse moves occur. This behavior may not be appropriate if your application requires all mouse move events. Handwriting recognition software is an example of this.

Included in your PersonalJava Solution for OS-9 package is the example Java source file `SimpleEventQueue.java`. When compiled, this source file produces two classes, `SimpleEventQueue.class` and `circularEventArray.class`. These classes override the default behavior of `EventQueue` for posting mouse events. All mouse events are posted in the system.

Note

There are two difficulties in overriding `EventQueue`. First, the queue data member is private. This prevents an application from appending events to the queue. Second, overriding all member functions does not work unless this class is in the package `java.awt`. The methods `removeSourceEvents()` and `changeKeyEventFocus()` are not public or protected, so they can only be overridden within the same package. This class stores all mouse events after one is posted into the super class. Therefore, no mouse movement data is lost from the system.

Contents of this Appendix

This appendix is made up of the following sections:

- **The SimpleEventQueue Example Java Program** provides sample code to produce the `SimpleEventQueue` and `CircularEventArray` classes.
- **Using the SimpleEventQueue Example Java classes** is a tutorial for using the `SimpleEventQueue` and `CircularEventArray` classes.

The SimpleEventQueue Example Java Program

When compiled, this example Java source code produces the `SimpleEventQueue` and `CircularEventArray` classes. The source for this example has been installed on the Windows development machine in the `E:\MWOS\SRC\PJAVA\EXAMPLES\ MOUSEMOVE` directory.

```
/*
** Overrides default behavior of EventQueue for posting mouse events. In
** this class, all mouse events are posted into the system. (EventQueue.java
** compresses mouse events where the last position is posted if multiple
** moves occur.)
**
** Note: There were two difficulties in overriding EventQueue. First, the
** queue data member is private, so we were unable to append all mouse events
** to the queue ourselves. Second, overriding all member functions won't work
** unless this class is in the package java.awt. (The methods
** removeSourceEvents() and changeKeyEventFocus() are not public or protected,
** so they can only be overridden within the same package.)
**
** This class will store all mouse events after one is posted into the super
** class. Thus, no mouse movement data is lost from the system.
**
*/

import java.awt.Event;
import java.awt.AWTEvent;
import java.awt.EventQueue;

public class SimpleEventQueue extends EventQueue {


  CircularEventArray _savedEvents;
  boolean _inList;  /* true if mouse event in super().queue */

  public SimpleEventQueue()
  {
        super();

        _savedEvents = new CircularEventArray();
        _inList = false;

        /* System.out.println("SimpleEventQueue"); */
  }

  /* overridden from EventQueue */
  public synchronized void postEvent(AWTEvent ev)
  {
```

```
        int id;
        id = ev.getID();

        if ((id == Event.MOUSE_MOVE) ||
                 (id == Event.MOUSE_DRAG))
          {
                if (_inList == true)
                  {
                        /* save event for later */
                        _savedEvents.add(ev);
                  }
                else
                  {
                        /* no events in super, so post it */
                        _inList = true;
                        super.postEvent(ev);
                  }
          }
        else
          /* not a mouse event, post it */
          super.postEvent(ev);
  }

  /* overridden from EventQueue */
  public synchronized AWTEvent getNextEvent() throws InterruptedException
  {
        AWTEvent ev = super.getNextEvent();
        int id = ev.getID();

        if ((id == Event.MOUSE_MOVE) ||
                 (id == Event.MOUSE_DRAG))
          {
                AWTEvent nextEvent = _savedEvents.remove();

                if (nextEvent != null)
                  /* post next event */
                  super.postEvent(nextEvent);
                else
                  /* no new events to post */
                  _inList = false;
          }

        return ev;
  }
}


/* this is a simple circular-array implementation of a queue. This will
** minimize memory requirements as compared to a linked list (since
** the garbage collector must run to reclaim nodes). Also, this is
** more efficient than Vector since a removal from the front won't
** move any other items in the array. */
class CircularEventArray {
  protected AWTEvent[] _data;
```

```
protected int _size;         /* number of slots in data */
protected int _front;        /* front index */
protected int _back;         /* back index */

/* NOTE:
** list is empty if front = back
** list is full if (back + 1) = front
**   (one dead cell is maintained to avoid confusion
**    between empty and full list)
** back points to the next unused cell
** front points to first valid cell (if not empty)
*/

protected final int _allocIncrement=20;


public CircularEventArray()
{
      _data = new AWTEvent[_allocIncrement];
      _size = _allocIncrement;
      _front = 0;
      _back = 0;
}

protected boolean full()
{
      int backAdj = _back + 1;

      if (backAdj >= _size)
        backAdj = 0;

      /* full? */
      if (backAdj == _front)
        return true;
      else
        return false;
}

public void add(AWTEvent ev)
{
      if (full())
        upsize();

      /* add item to back */
      _data[_back] = ev;

      _back++;
      if (_back >= _size)
        _back = 0;
}

public AWTEvent remove()
{
      AWTEvent ev;
```

```
        /* check for empty list */
        if (_back == _front)
          ev = null;
        else
          {
                /* remove front item */
                ev = _data[_front];
                _data[_front] = null;

                _front ++;
                if (_front >= _size)
                  _front = 0;
          }

        return ev;
  }

  protected void upsize()
  {
        int src=_front;
        int dest=0;
        AWTEvent[] newArray = new AWTEvent[_size + _allocIncrement];
        AWTEvent cur;

        /* copy items from original array to newArray */
        cur = _data[src];
        while (cur != null)
          {
                /* copy data */
                newArray[dest] = cur;
                _data[src] = null;

                /* setup for next iteration */
                dest++;
                src++;
                if (src >= _size)
                  src = 0;

                cur = _data[src];
          }

        _data = newArray;
        _front = 0;
        _back = dest;
        _size = _size + _allocIncrement;

        /* System.out.println("CircularArray upsize to " + _size); */
  }
}
```

Using the SimpleEventQueue Example Java classes

The following Java source file can be compiled to generate the `SimpleEventQueue` and `CircularEventArray` classes. These classes can be used to retain all mouse move messages received from the system.

The SimpleEventQueue Example Java Source File

The source for this example has been installed on the Windows development machine in the `E:\MóWOS\SRC\PJAVA\EXAMPLES\MOUSEMOVE` directory.

Compiling the Source File

In a DOS shell on a Windows development machine, compile the source file using the Java compiler:

```
> cd \MWOS\SRC\PJAVA\EXAMPLES\MOUSEMOVE
> javac SimpleEventQueue.java
```

Once the compilation succeeds, the following class files are created in the same directory as the Java source file:

```
SimpleEventQueue.class
CircularEventArray.class
```

Note

If the compilation fails, make sure you typed in and named the program exactly as shown above. Capitalization is important.

Enhancing the awt.properties file

To make the JVM (Java Virtual Machine) use the `SimpleEventQueue` and `CircularEventArray` classes, you must enhance the `awt.properties` file.

Diskless System

If your OS-9 target machine is a diskless system complete the following steps:

Step 1. Change to the properties files directory on the Windows machine:

```
cd \MWOS\SRC\PJAVA\LIB
```

Step 2. Add the following line to the `awt.properties` file:

```
AWT.EventQueueClass=SimpleEventQueue
```

The new `awt.properties` specifying the `SimpleEventQueue` as the `EventQueueClass` is added to `pjava_home.mar` the next time `pjruntime` is generated.

For More Information

For a complete discussion of `pjavamods`, refer to **Chapter 4: Choosing a PersonalJava Diskless Strategy**.

Disk-Based System

If your OS-9 target machine is a disk-based system and loads the JVM and its resources from a MWOS directory on a system disk, complete the following steps:

Step 1. Change to the properties files directory on the OS-9 machine:

```
cd /h0/MWOS/SRC/PJAVA/LIB
```

Step 2. Add the following line to the `awt.properties` file:

```
AWT.EventQueueClass=SimpleEventQueue
```

Transferring the SimpleEventQueue Classes to the Target OS-9 System

The `SimpleEventQueue.class` and `CircularEventArray.class` must be put in your classpath on the Target OS-9 System.

Diskless System

If your OS-9 target machine is a diskless system, the `SimpleEventQueue` and `CircularEventArray` classes can either be pre-loaded into `libclasses.so` using the JCC (JavaCodeCompact) or changed into data modules.

For More Information

Refer to **Chapter 4: Choosing a PersonalJava Diskless Strategy** for a complete discussion on adding your classes to a diskless OS-9 Target.

Disk-Based System

If your OS-9 target machine is a disk-based system and loads the JVM from a `MWOS` directory on a system disk, the `SimpleEventQueue` and `CircularEventArray` classes can be transferred to the OS-9 target machine using FTP.

For More Information

Refer to the section **Tips for Running Your Application or Applet** in **Chapter 3: Creating Java Applications for OS-9** regarding enhancing the `CLASSPATH` environment variable so the JVM can find additional classes.

Appendix D: Microware Archive Tool

PersonalJava Solution for OS-9 includes a new utility for creating and extracting archives: Microware Archive Tool (MAT). MAT is shipped as a Windows hosted utility (MWOS\DOS\BIN\mat.exe) and an OS-9 hosted utility (MWOS\OS9000\<proc>\CMDS\mat).

The port-specific makefile in MWOS\OS9000\<portproc>\PORTS\<port>\PJAVA\TARGET uses MAT to create an archive that can be extracted on your target. MAT automatically handles the line ending translations necessary when moving text files from Windows to OS-9.

This appendix includes the full documentation for MAT.

Usage

MAT's operations are very similar to Unix's tar command. It can create, list the contents of, and extract archives. It is generally used to create archives on the development host for extraction on the development target.

Archive Creation

MAT walks the directories specified on the command line and adds each file they contain to the archive. Empty directories are also stored in the MAT archive and will be re-created upon extraction. Text files are stored in the archive in an line ending independent format. Multiple MAT archives can be merged (e.g. os9merge) together to form a larger MAT archive.

Archive Contents Listing

MAT can list the contents of a MAT archive. The pathlist of each file is printed. If needed, the permissions, modification date, and file size can also be printed.

Archive Extraction

MAT extracts each file and empty directory from MAT archives to the same relative location that it appeared during archive creation. The modification date and, optionally, the permissions of each file are restored.

During extraction, text files' line endings are written appropriately for the default host platform. The default host platform is the platform on which MAT is currently running. That is, an extraction done on Windows will yield Windows' line endings (CR LF). The default line endings can be over-ridden (see -t).

Command-line

The MAT command line consists of options and arguments. Options begin with minus (‘-’) and control how MAT uses the arguments. The options and arguments can appear in any order. See the option descriptions for what the arguments mean in the various modes.

The following are the command line options available for MAT:

`-?` — `print usage information`

specifies MAT should print the usage and command-line option summary and exit

`-a` — `extract file names as contained in the archive`

specifies that the extracted file name should be exactly as specified in the archive. MAT normally translates illegal pathname characters and shortens pathnames that are too long. [OS-9 resident version only]

`-b[=]<size>[k|K]`

`specify the I/O buffer size (default = 128K)`

specifies the number of kilobytes of RAM that MAT should use when copying files into or out of the archive

`-c` — `create mat archive (default directory = .)`

specifies MAT should create archives for all the directories specified in the arguments

If no arguments are used, the default directory to archive will be ‘.’, the current directory.

`-e` — `extended listing or extended verbose information`

specifies additional information is desired

When creating, listing, or extracting an archive (see -c, -l, -x) the permissions, modification date, and size are printed for each file.

-f `force overwrite on read-only files`
specifies that MAT should overwrite existing destination files, adding write permission to read-only files if necessary

-l `list contents of mat archive`
specifies that MAT should display the contents of all the archives specified in the arguments. Use -e for extended information.

-o[=]<file> `specify output file for create operation`
specifies the file that MAT should store the archive in

This option is required when -c is used. If <file> exists, it is truncated and overwritten.

-p `preserve permissions during extract`
specifies that MAT should preserve the original file permissions during extraction

By default, MAT will add write permission for all users that have read permission.

-r `overwrite existing destination files`
specifies that MAT should overwrite existing destination files

Use -f to force overwrite of read-only files.

-t[=]<d|o|u> `translate ASCII file EOLs to (D)OS, (O)S-9, or (U)nix when extracting (default = DOS)`
specifies the desired line ending for text files.

-t only needs to be used when the desired line ending differs from the normal line ending on the host.

-v `verbose create or extract operation`
specifies that MAT should print information about its progress

If -v is used during creation or extraction MAT will print each directory and file encountered. -e can be used to get extended information.

`-w`	`generate only warnings applying file characteristics during extraction` specifies that MAT shouldn't exit if it has trouble setting the permissions or modification dates on created files
`-x`	`extract contents of mat archive` specifies that MAT should extract the contents of each MAT archive specified as command-line arguments
`-z[[=]<file>]`	`read additional command line arguments from <file> (default = standard input)` specifies that MAT should read additional command line arguments and/or options from the specified file or standard input if no file is specified

Examples

To create an archive of everything below the current directory:

```
$ mat -cvo=../simple.mat
OBJ
OBJ/libjckjni.so
OBJ/STB
OBJ/STB/libjckjni.so.map
OBJ/STB/libjckjni.so.stb
RELS
RELS/cmp.r
RELS/cvt.r
RELS/jckjni.r
RELS/libtable.r
```

To list the extended contents of an archive, complete the following:

```
$ mat -le simple.mat
  Perms       Modified       Size  Name
--------- ----------------- ------ --------------------------------------------
                                   OBJ
xwrxwrxwr 99/10/21 09:37:36 743312 OBJ/libjckjni.so
                                   OBJ/STB
xwrxwrxwr 99/10/21 09:37:36  82531 OBJ/STB/libjckjni.so.map
xwrxwrxwr 99/10/21 09:37:36  67696 OBJ/STB/libjckjni.so.stb
                                   RELS
xwrxwrxwr 99/10/21 09:30:07    806 RELS/cmp.r
xwrxwrxwr 99/10/21 09:30:08    414 RELS/cvt.r
xwrxwrxwr 99/10/21 09:37:28 744951 RELS/jckjni.r
xwrxwrxwr 99/10/21 09:37:33 134902 RELS/libtable.r
```

To extract an archive, complete the following:

```
$ mat -xv simple.mat
OBJ
OBJ/libjckjni.so
OBJ/STB
OBJ/STB/libjckjni.so.map
OBJ/STB/libjckjni.so.stb
RELS
RELS/cmp.r
RELS/cvt.r
RELS/jckjni.r
RELS/libtable.r
```

Appendix E: Sources of Information

This appendix provides a bibliography of sources available for programming in Java.

RadiSys.

MICROWARE SOFTWARE

Sources

Arnold, Ken, and James Gosling. *The Java Programming Language.* Addison-Wesley Pub Co. 1996.

Chan, Patrick, and Rosanna Lee. *The Java Class Libraries: An Annotated Reference*. Addison-Wesley Pub Co. 1997.

Flanagan, David. *Java in a Nutshell*. O'Reilly & Associates, Inc. 1996.

Gosling, James, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Pub Co. 1996.

Gosling, James, and Frank Yellin. *The Java Application Programming Interface, Volume 1*. Longman Pub Group. 1996.

Gosling, James, and Frank Yellin. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets*. Addison-Wesley Pub Co. 1996.

Jackson, Jerry R., and Alan L. McClellan. *JAVA by Example*. SunSoft Press. 1996

Lea, Doug. *Concurrent Programming in Java*. Addison-Wesley Pub Co. 1996.

Lindholm, Tim, and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Pub Co. 1997.

Glossary

This glossary defines terms found in this manual.

For More Information

The terms listed below are common to this manual. For additional terms, refer to the ***Glossary of Java and Related Terms*** found at `http://java.sun.com`.

Abstract Window Toolkit

A device-independent layer of the Java API. It enables Java applications to create windows, buttons, menus, and a variety of other typical windowing widgets.

API

Application Programming Interface.

Applet

A Java program that executes in the context of a web browser capable of executing that program or the Sun Appletviewer.

Application

A Java program that does not need a web browser or appletviewer for execution. A web browser is a Java application.

Application Programming Interface

The specification of how a programmer writing an application accesses the behavior and state of classes and objects.

AWT

Abstract Window Toolkit.

Class

The fundamental structure in Java. It corresponds to a struct in C.

EmbeddedJava Platform

The EmbeddedJava platform consists of core and standard extension APIs, and is designed specifically for severely resource-constrained environments. The EmbeddedJava platform includes a feature level subset of Java and is used to run a variety of products including mobile phones, pagers, process control, instrumentation, office peripherals, and networking routers and switches. See also **PersonalJava Platform** Platform.

Garbage Collection

The detection and freeing of memory that is no longer in use. The Java runtime system performs garbage collection so programmers never explicitly free objects.

Glue Code

Code that connects two component blocks.

Heap

A block of memory.

HTML

HyperText Markup Language.

HTTP

HyperText Transfer Protocol.

HyperText Markup Language

This is a file format, based on SGML, for hypertext documents on the Internet. It is very simple and allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs.

HyperText Transfer Protocol

The Internet protocol, based on **TCP/IP**, used to fetch hypertext objects from remote hosts.

JAR

Java Archive.

Java

A general purpose object-oriented programming language.

Java Archive

A file format based on the popular ZIP file format and used for aggregating many files into one. Although JAR can be used as a general archiving tool, it was developed so Java applets and their requisite components (`.class` files, images, and sounds) can be downloaded to a browser in a single HTTP transaction, rather than opening a new connection for each piece.

Java Development Kit

The Java programming language is available as a product known as the Java Development Kit (JDK).

Each release of the JDK contains:

- Java Compiler
- Java Virtual Machine
- Java Class Libraries
- Java Applet Viewer
- Java Debugger and other tools

Java Linker

A module that builds an executable, complete program from component machine code modules. The Java linker creates a runnable program from compiled classes.

Java Native Interface

The JNI is a native programming interface. It allows Java code that runs inside a JVM to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. The most important benefit of the JNI is it imposes no restrictions on the implementation of the underlying JVM. Therefore, JVM vendors can add support for the JNI without affecting other parts of the JVM. Programmers can write one version of a native application or library and expect it to work with all JVMs supporting the JNI.

Java Virtual Machine

An abstract specification for a computing device that can be implemented in different ways, in software or hardware. You compile to the instruction set of a virtual machine much like you would compile to the instruction set of a microprocessor. The Java Virtual Machine consists of a bytecode instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods.

JDK

Java Development Kit.

JNI

Java Native Interface.

JVM

Java Virtual Machine.

MAUI

Multimedia Application User Interface.

Methods

Functions that act on the instance variables of a class.

Module

A separately compiled or assembled piece of code with well-defined functionality and a well-defined interface. Modular design is an integral concept of state-of-the-art structured programming techniques. A work file is usually a source code module.

Multimedia Application User Interface

A shared library for managing the display graphics, text, and sounds using the multimedia file manager (MFM).

Native Method

Code that allows Java, which is platform-independent, to interact with platform-specific code.

NC

Network Computer.

Network Computer

A computer with minimal memory, disk storage, and processor power designed to connect to a network, especially the Internet.

OS-9

An operating system created by RadiSys Microware Communications Software Division, Inc.

PersonalJava Platform

The PersonalJava platform consists of core and standard extension (optional) APIs. It is designed specifically for resource-limited environments, with the addition of specific features required by consumer applications. Applications in development for PersonalJava platforms include hand-held computers, set-top boxes, game consoles, mobile hand-held devices, and smart phones. See also **EmbeddedJava Platform** Platform.

Pipe

A special type of I/O path that connects and synchronizes the standard output of a program to the standard input of another simultaneously running program. A pipe is a first-in, first-out (FIFO) buffer and contains 128 bytes, unless a different buffer size is specified when the pipe is created. Chains of piped programs are called pipelines.

PPP

Point-to-Point Protocol.

Profile

Subset of the Java classes created to make Java function more efficiently.

Random Block File Manager

The OS-9 file manager module supporting random-access, block-oriented mass storage devices (such as disk systems). RBF can handle any number or type of such systems simultaneously. It is responsible for maintaining the logical and physical file structure of OS-9.

RBF

Random Block File Manager.

RTOS

Real-Time Operating System.

SCF

Sequential Character File Manager.

Sequential Character File Manager

The OS-9 file manager module supporting sequential-access, character-oriented devices such as terminals and printers or RS-232 ports or pointer devices (game pads and infrared remote controls). SCF can handle any number or type of such systems simultaneously.

Serial Line Internet Protocol

Popular form of encapsulation for IP datagrams. Specified in RFC1055, it has become popular with the use of high-speed modems and RS-232 serial ports found on PCs.

Set Top Box

A consumer electronic device that connects to a television and to a video-on-demand network by coaxial cable, phone line, or fiber optic network.

SLIP/CSLIP

Serial Line Internet Protocol/Compressed Serial Line Internet Protocol.

SPF

Stacked Protocol File Manager.

Stacked Protocol File Manager

The Stacked Protocol File manager supports asynchronous delivery of network data to applications and hardware devices.

STB

Set Top Box.

Super User

A super user has a group.user ID of 0.0. A super user can access and manipulate any file or directory on the system regardless of the file's ownership.

TCP/IP

Transmission Control Protocol/Internet Protocol.

Thread

A Java class allowing concurrent applications performing multiple simultaneous activities to run.

UDP

User Data Packet.

User Data Packet

Packet-based network protocol.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:______________________________________

Phone:____

Fax:________________________Email:____________________

Product Name:

Description of Problem:

Host Platform____________________________________

Target Platform___________________________________

RadiSys.
MICROWARE SOFTWARE