Home

Using JavaCodeCompact for OS-9®

Version 3.1

RadiSys
THE POWER OF WE

www.radisys.com
Revision C • July 2006

Copyright and publication information

Disclaimer

Reproduction notice

Table of Contents

Chapter 1: Pre-loading Classes with JavaCodeCompact 5

6 Pre-Loading Class Files Overview
6 libclasses.so Module
8 The Java Pre-loader Application
8 Syntax
8 Description
9 Java Options
9 Options
13 Building a Shared Class Library Using JavaCodeCompact
13 Stage 1. Compiling the Java Classes
13 Stage 2. Pre-loading the Java Class Files
14 Stage 3. Assemble and Link the Pre-loader Output
16 Using the Pre-loaded Classes
16 Using Pre-loaded Classes from the Boot File
17 Using Pre-loaded Classes from RAM

Chapter 1: Pre-loading Classes with JavaCodeCompact

PersonalJava™ Solution for OS-9® enables Java class files to be pre-loaded—converted from the class file format into data structures used by the Java Virtual Machine (JVM).

This chapter explains how to generate pre-loaded classes using the host-based JavaCodeCompact (JCC) tool. It includes the following sections:

- **Pre-Loading Class Files Overview**
- **The Java Pre-loader Application**
- **Building a Shared Class Library Using JavaCodeCompact**
- **Using the Pre-loaded Classes**

Pre-Loading Class Files Overview

Pre-loaded Java class files can be used in PersonalJava™ Solution for OS-9. Pre-loading enables class file information to be loaded into memory in a ready-to-use modular format instead of having to be loaded from a disk or over a network. In addition, using pre-loaded classes offers benefits for size and speed.

Pre-loaded classes are usually about the same size as classes in an uncompressed .zip file. Normally, when classes are loaded from a .zip file into the JVM, they expand roughly by a factor of two. Therefore, using pre-loaded classes results in about a fifty percent memory savings. Also, because pre-loaded classes can reside in ROM memory, the total system RAM requirement is reduced.

Loading and verifying classes takes a significant amount of processing time. Since pre-loaded classes do not require this operation, JVM start time is reduced.

libclasses.so Module

Pre-loaded classes on OS-9 systems are contained in a module called `libclasses.so`.

At start-up, the JVM searches the module directory and disk directories specified in your `LD_LIBRARY_PATH` environment variable for a module with this name. The debug version of the JVM (`pjava_g`) looks for `libclasses_g.so` (see -g option). If the file is found and in the proper format, the JVM uses the pre-loaded classes found in that module. Because `libclasses.so` contains data structures with absolute pointers, it must be created to exist at a fixed address. During the pre-load process, the address of the `libclasses.so` module is given to the pre-loader tool as one of its parameters.

In order for the pre-loader to create a ROMable image, it must be able to resolve all references made in the class files being processed. If the pre-loader does not find a referenced class among the list of classes being operated on, it prints a warning message. You must modify the list of classes being operated on by the pre-loader to eliminate these warning messages.

The Java Pre-loader Application

This section describes the JavaCodeCompact pre-loader tool and the options you can use when running it.

Syntax

```
java [Java options] JavaCodeCompact [options]
filename ...
```

Description

JavaCodeCompact combines one or more Java class files and produces a target-system dependent assembly source file. This file contains the given classes in a preloaded format that can be assembled and linked into a shared library. All of these operations are performed on the host development system.

Linking Java Programs

Here is an outline of the conventional mechanism for class loading:

- use `javac` to compile Java source files into Java class files
- load the class files into a Java system, either individually or as part of `jar` archive files
- resolve references to other class definitions upon demand

JavaCodeCompact provides an alternate means of program linking and symbol resolution—one that provides a less-flexible model of program building, but which helps minimize the initialization time and memory requirements of the JVM.

JavaCodeCompact can perform the following functions:

- combine multiple input files by combining much of their symbolic information into a shared string and constant pool and concatenating other parts of the classes' definitions
- determine an object instance's layout and size
- determine the layout of an object's method table
- change the representation of some Java byte codes to their quick forms

Java Options

`-mx20M`	Increases the heap to 20MB This program often has to be run with an increased maximum heap size.

Options

`filename`	designates the name of a file used as input The contents should be included in the output. File names are not modified by any pathname calculus. File names with a `.class` suffix are read as single class files. File names with `.jar` or `.zip` suffixes read as Zip files. These Zip files must contain only class files as elements.
`-o outfilename`	designates the name of the output file to be produced Conventionally, the file name ends with a suffix of .a for assembly-language output. This is not critical to the operation of the program. In the absence of this option, a file is produced with a name based on the name of the first input file. The name is stripped of pathname prefix and any suffix and has .a appended.

`-q`	enables transformation of method code to its quickened form
	Quickened refers to an optimization implemented in Sun's version of the JVM. Refer to ***The Java Virtual Machine Specification*** for a complete discussion of this optimization.
	Many Java bytecode instructions refer to symbolic quantities such as the offset of a field or of a method, or simply to the name of a type. Normally, the JVM resolves these references when it first executes the instruction. It then re-writes the instruction in place. This procedure yields code that cannot be placed in ROM.
	Java bytecodes that are resolved and quickened at link time are read-only and can be placed in ROM. Any instructions that refer to symbols that are not resolved are not quickened.

Note

The `-q` option should only be used on the final link step.

`-qlossless`	identical to `-q`, but leaves the resulting bytecode amenable to Just-In-Time (JIT) compilation
`-c`	cumulative linking
	Classes unresolved by the linking of class files explicitly listed as linker arguments are searched for using the `-classpath` option, and linked as they are found. File names are formed by concatenating a path prefix, the

	character `java.io.File.separatorChar` (on Windows, a \), the name of the class being sought, and the suffix `.class`.
`-classpath path`	specifies the path JCC uses to look up classes Directories and Zip files are separated by `java.io.File.pathSeparatorChar`, which on Windows is a semi-colon. Multiple classpath options are cumulative, and searched left-to-right. This option is only used in conjunction with the `-c` cumulative-linking option.
`-v`	turns up the verbosity of the linking process This option is cumulative. Currently up to three levels of verbosity are understood. This option is typically used as a debugging aid.
`-f filename`	read options and class file names from the specified file
`-g`	enables the writing of line-number tables, source files names in the output, and a local variable table The information must be available in the input data, as it is with normal class files. These tables are not written by default. This option also suppresses the code inlining optimization. This options should be used if pre-loaded classes are desired when using the debug version of the JVM, pjava_g. The debug version of the JVM can not use pre-loaded classes that were built without -g, nor can the optimized version of the JVM use pre-loaded classes that were built with -g.
`-jniClass class`	specifies that the data structures pre-loaded for `class` should use the Java Native Interface (JNI) method interface

`-arch targetarchitecture`

Designates the assembly language used in writing the output

The argument is case insensitive. Use only when an assembly language output file is to be produced. The currently supported options include the following:

- ARM—generate StrongARM assembly code files
- PPC—generate PowerPC assembly code files.
- SH—generate SuperH-3/SuperH-4 assembly code files.
- X86—generate x86 assembly code files.

`-imageAttribute moduleBase=<module address>`

specifies the address where the resulting shared library is loaded

In order for the pre-loaded classes to be used by the JVM, the shared library must be located at the address specified by this option. Unlike most OS-9 modules, the pre-loaded classes shared library is not position independent. The address can be specified in either hexadecimal (0x prefix) or decimal form.

Building a Shared Class Library Using JavaCodeCompact

Following is the procedure for building `libclasses.so`, the pre-loaded classes file. `libclasses.so` can be used from ROM or RAM.

Stage 1. Compiling the Java Classes

Step 1. Use the `javac` command to convert Java program source files into Java class files.

The standard file name convention is as follows:

`<classname>.class`

Step 2. Load the class files, either individually or as part of .zip or .jar archive files into a Java system.

Their references to other class definitions are resolved upon demand by the class loading and resolving mechanism specified as part of the Java language semantics.

Stage 2. Pre-loading the Java Class Files

Step 1. From your Windows host development system, pre-load the classes that you compiled in Stage 1 using the following:

- `The -q` option builds an image of the Java Development Kit (JDK) internal data structures representing all the included classes in the form of an assembler input file.
- `The -imageAttribute moduleBase=<module address>` option indicates where the resulting shared library object is loaded.

For More Information

Refer to the **Using the Pre-loaded Classes** section for more information about the module address.

- The `-arch <ARCH>` option specifies the target processor architecture to be used. The following definitions of <ARCH> are currently supported: `ARM` for StrongARM, `PPC` for PowerPC, `SH` for SH-3/SH-4, and `X86` for Intel x86.
- The `-c` option specifies cumulative linking.
- The `-classpath` option specifies the path for JavaCodeCompact to look up classes during cumulative linking.

Example

Following is a typical class file pre-loading command line:

```
java -mx20m JavaCodeCompact -q -imageAttribute
moduleBase=0x30000000 -arch ARM -c -jniClass
java.lang.SecurityManager -jniClass java.lang.Thread
-jniClass java.security.AccessController -classpath
c:\MWOS\SRC\PJAVA\LIB\classes.zip
-f classes.lst -o classes.a
```

Stage 3. Assemble and Link the Pre-loader Output

Step 1. Assemble and link the pre-loader output to produce an OS-9 shared library named either `libclasses.so` or `libclasses_g.so`.

The name depends on whether you are using the debugging or non-debugging version of the JVM.

Example

Following is a typical assemble and link command line:

```
xcc -olg -tp=armv4 -r -l=libsm.l -l=cpu.l -cs=smstart.r
classes.a -fd=libclasses.so
```

In this example, `classes.a` is the assembly output from JavaCodeCompact.

Using the Pre-loaded Classes

Once `libclasses.so` is created with JavaCodeCompact, you can use the pre-loaded classes either from ROM or RAM. The following procedures describe how to use the `libclasses.so` file in each particular situation.

Using Pre-loaded Classes from the Boot File

Complete the following steps to use your pre-loaded classes from boot file:

Step 1. Build libclasses.so from `MWOS\OS9000\<portproc>\PORTS\<portname>\PJAVA\JCC` with the default module address in the make file.

Step 2. Put this version of `libclasses.so` in your boot image as a placeholder. This placeholder will help you determine the correct start address for the `libclasses.so` module.

Step 3. Create the boot image and use it to boot the system.

Note the address of `libclasses.so` in memory with `mdir -e`, you will need this information in the next step.

Step 4. Edit the make file in the `JCC` directory and change the address to match the address recorded from the previous step.

Invoke the make file again to rebuild `libclasses.so` with the correct address.

Step 5. Copy the `libclasses.so` module into your boot-build location, and build a new boot image. The result is a `libclasses.so` that will be found at the correct address.

Using Pre-loaded Classes from RAM

Complete the following steps to use your pre-loaded classes from RAM:

Step 1. Create a 2.5MB plane of colored memory on the target machine by adding an entry in the colored memory definition list. This becomes part of your init module.

Note

For a one-time change, use editmod. For a permanent change, modify the configuration source files and rebuild the init module.

For More Information

Refer to the ***OS-9 Configuration Reference*** manual for information about colored memory in OS-9.

Step 2. Build a shared class library using JavaCodeCompact.

These steps are described in the **Building a Shared Class Library Using JavaCodeCompact** section. Use the address of your colored memory plane as the base address for the library. You can proceed through the steps manually or invoke a makefile for this purpose.

Step 3. Pad the `libclasses.so` file to 2.5Mb so the module begins exactly at the start of the colored region. For example, type the following on your Windows host system: `padrom 0x280000 libclasses.so`.

Step 4. Move the library to your target machine, once it has been created, and load it into RAM using a command such as the following:

```
load -ldc=0x90 libclasses.so
```

In this example, 0x90 specifies the memory color you created. The result is a pre-loaded class library found at a known address.