KVM Porting Guide

CLDC, Version 1.1

Java™ 2 Platform, Micro Edition

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A. 650-960-1300

March, 2003

Please
Recycle

Contents

Figures

Tables

CHAPTER 1

About This Document

This document provides information for porting the K Virtual Machine (KVM), version 1.1, to a new platform. KVM is a Java Virtual Machine implementation that is commonly used as the execution engine for J2ME CLDC (Java™ 2 Micro Edition, Connected Limited Device Configuration.)

1.1 Who should use this document

This document is intended primarily to those individuals and companies who want to port Sun's reference implementation of the K Virtual Machine to a new platform. The document is useful also to those persons who want to learn more about the internal details of the KVM.

1.2 Related documentation

The Java™ Language Specification (Java Series), Second Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3

Programming Wireless Devices with the Java™ 2 Platform, Micro Edition (Java Series) by Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. Addison-Wesley, 2001, ISBN 0-201-74627-1 (Second edition planned for publication in Q2, 2003.)

Connected, Limited Device Configuration Specification, version 1.1, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/jsr/detail/139.jsp`

Connected, Limited Device Configuration Specification, version 1.0, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html`

Mobile Information Device Profile Specification, version 2.0, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/jsr/detail/118.jsp`

Mobile Information Device Profile Specification, version 1.0, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html`

Java 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices, A White Paper, Sun Microsystems, Inc.
`http://java.sun.com/products/cldc/wp/KVMwp.pdf`

KVM Debug Wire Protocol (KDWP) Specification, Sun Microsystems, Inc.

K Native Interface (KNI) Specification, version 1.0, Sun Microsystems, Inc.

IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985), American National Standards Institute

Making Java Work for High-End Computing, Java Grande Forum Report

Improving Java for Numerical Computation, Numerics Working Group, Java Grande Forum

CHAPTER 2

Introduction to KVM

2.1 K Virtual Machine (KVM)

KVM (also known as the *K Virtual Machine* or the *KJava Virtual Machine*) is a compact, portable Java™ virtual machine that has been designed specifically for small, resource-constrained devices such as cellular phones, pagers, personal organizers, mobile Internet devices, point-of-sale terminals, home appliances, and so forth.

The high-level design goal for the KVM team was to create the smallest possible "complete" Java virtual machine that would maintain all the central aspects of the Java programming language, and that would nevertheless run in a resource-constrained device with only a few tens or hundreds of kilobytes of available memory (hence the name K, for kilobytes). More specifically, KVM is designed to be

- small, with a static memory footprint of the virtual machine core starting from about 50-70 kilobytes (depending on the target platform and compilation options),
- clean and highly portable,
- modular and customizable,
- as "complete" and "fast" as possible without sacrificing the other design goals.

KVM is implemented in the C programming language, so it can easily be ported onto various platforms for which an ANSI C compiler is available. The virtual machine has been built around a straightforward bytecode interpreter with various compile-time flags and options for helping porting efforts and space optimization.

KVM has been developed as part of a larger effort to provide a modular, scalable architecture for the development and deployment of portable, dynamically downloadable and secure applications in consumer and embedded devices. This larger effort is called the *Java 2 Micro Edition* (also known as Java 2 ME or J2ME).

The K Virtual Machine is typically used as the implementation-level foundation for the following J2ME technology standards: *Connected, Limited Device Configuration* (CLDC) and *Mobile Information Device Profile* (MIDP). Further information on KVM, CLDC, MIDP and Java 2 Micro Edition in general is available in separate documents listed in Section 1.2 “Related documentation.”

KVM is derived from a research system called *Spotless* developed originally at Sun Microsystems Laboratories in early 1998. More information on the Spotless system is available in the Sun Labs technical report *The Spotless system: implementing a Java system for the Palm connected organizer*.

2.2 Differences between KVM 1.1 and KVM 1.0.3 / 4

KVM 1.1 is the first KVM implementation that supports the *CLDC Specification* version 1.1. This release implements all the new features that have been added in CLDC 1.1, including:

- Floating point support has been added.
  - All floating point byte codes are supported by CLDC 1.1.
  - Classes `Float` and `Double` have been added.
  - Various methods have been added to the other library classes to handle floating point values.
- Weak reference support (small subset of the J2SE weak reference classes) has been added.
- Classes `Calendar`, `Date` and `TimeZone` have been redesigned to be more J2SE-compliant.
- Error handling requirements have been clarified, and one new error class, `NoClassDefFoundError`, has been added.
- In CLDC 1.1, `Thread` objects have names, like threads in J2SE do. The method `Thread.getName()` has been introduced, and the `Thread` class has a few new constructors that have been inherited from J2SE.
- Various minor library changes and bug fixes, such as the addition of the following fields and methods:
  - `Boolean.TRUE` and `Boolean.FALSE`
  - `Date.toString()`
  - `Random.nextInt(int n)`
  - `String.intern()`
  - `String.equalsIgnoreCase()`
  - `Thread.interrupt()`

- Minimum total memory budget for CLDC has been raised from 160 to 192 kilobytes, mainly because of the added floating point functionality.
- Specification text tightened and obsolete subsections removed.
- Much more detailed verifier specification ("CLDC Byte Code Typechecker Specification") is provided as an appendix to the *CLDC Specification* version 1.1.

For a more detailed summary of the differences between CLDC 1.0 and 1.1, refer to the *CLDC Specification* version 1.1.

The KVM 1.1 release includes all the functionality and fixes of the previous KVM 1.0.4 release. In addition to the new features that are specific to *CLDC Specification* version 1.1, the main features of KVM 1.1 compared to KVM 1.0.3 include:

- Support for the K Native Interface (KNI).
- Rewritten class loader that can support error handling in a J2SE-compliant fashion.
- More portable runtime verifier implementation that is easier to port to other virtual machines.
- Various minor bug fixes and enhancements.

For most up-to-date information, refer to the release notes and KVM product website (`http://java.sun.com/products/kvm`).

Important. Unlike the earlier KVM/CLDC releases, the KVM 1.1 implementation no longer includes network protocol implementations, graphical user interface code, or other components that are outside the scope of *CLDC Specification*. The networking code as well as user interface components are provided in additional J2ME software releases such as the MIDP reference implementation.

CHAPTER 3

Compiler Requirements

In order to be able to compile the KVM codebase, you must have a C compiler capable of compiling ANSI-compliant C files. Your compiler must define the basic C types as shown below in Table 1.

TABLE 1 Basic types

Type	Description
`char`	An 8-bit quantity. It can be signed or unsigned.
signed `char`	A signed 8-bit quantity.
unsigned `char`	An unsigned 8-bit quantity.
`short`	A signed 16-bit quantity.
unsigned `short`	An unsigned 16-bit quantity.
`int`	A signed quantity. It is either 16 or 32 bits.
unsigned `int`	A unsigned quantity. It is either 16 or 32 bits.
`long`	A signed 32-bit quantity.
unsigned `long`	An unsigned 32-bit quantity.
`void *`	A 32-bit pointer.

If your J2ME configuration or profile supports floating point numbers, your compiler must support the floating point types shown below in Table 2.

TABLE 2 Floating point types

Type	Description
`float`	A 32-bit floating point value.
`double`	A 64-bit floating point value.

All KVM implementations support the Java type `long`.[1] It is preferable that your compiler support 64-bit integers; however this is not a requirement. Porting the Java type `long` is discussed in Chapter 9, “64-bit Support.”

Your compiler must have some means of indicating additional directories to be searched for “includes” of the form:

```
#include <filename>
```

Our reference implementation has only been tested on machines with 32-bit pointers and that do not require “far” pointers of any sort. We do not know if it will run successfully on platforms with pointers of other sizes.

The codebase has been successfully compiled with the following compilers:

- Sun C Compiler 5.0, 5.2 and 5.3 on Solaris,
- GNU C 2.91.66 (egcs-1.1.2) compiler on Red Hat Linux,
- GNU C 2.95.2 compiler on Solaris and Windows NT 4.0,
- Microsoft Visual C++ 6.0 Professional on Windows NT 4.0 and Windows 2000.

The only non-ANSI C feature in the KVM source code base is its use of 64-bit integer arithmetic. Refer to Chapter 9 for further information on 64-bit support.

1. Note that in the Java programming language, the type `long` is always 64 bits. TABLE 1 on page 7 assumes that, as in most current C implementations, the type `long` represents a 32-bit quantity. This document uses the phrase “The Java type `long`” to refer to the 64-bit meaning.

CHAPTER 4

Required Port-Specific Files and Functions

This section describes those files and functions that must be defined for each port.

4.1 File `machine_md.h`

Every KVM port must provide a file named `Vm`*`Port`*`/h/machine_md.h`. The purpose of this file is to override the default compile time definitions and declarations provided in `VmCommon/h/main.h`, and supply any additional definitions and declarations that your specific platform might need. See Chapter 6, "Compilation Flags, Definitions and Macros" for a list of the definitions and declarations that your port will often need to override.

All port-specific declarations, function prototypes, `typedef` statements, `#include` statements, and `#define` statements must appear either in this `machine_md.h`, in a file included directly or indirectly by `machine_md.h`, in some file automatically included by your development environment,[1] or via compiler switches.[2]

Port-specific functions can appear in any machine-specific file. Unless otherwise stated, any required port-specific function can also be defined as a macro, provided that its implementation is careful to ensure that each argument is evaluated exactly once.

1. Metrowerks CodeWarrior, for example, allows the user to create a *prefix file.*

2. Some compilers allow you to add the switch `-Dname=value`, which is equivalent to putting
   `#define name value`
   at the start of the file.

4.2 File `main.c`

You will generally need to provide a new version of `main.c` that is suitable for your target platform. The default implementation provided in directory `VmExtra/src/main.c` can be used as a starting point for platform-specific implementations. Refer to Chapter 7, "Virtual Machine Startup," for further information.

4.3 Runtime functions that require porting efforts

Each port must define the functions given below (see `VmCommon/h/runtime.h`). They may be defined as either macros or as C code. Traditionally, the C code is placed in a file named `VmPort/src/runtime_md.c`

- `void AlertUser(const char* message)`
  Alert the user that something serious has happened. This function call usually precedes a fatal error.
- `cell *allocateHeap(long *sizeptr, void **realresultptr)`
  Create a heap whose size (in bytes) is approximately the long value `*sizeptr`. The heap must begin at an address that is a multiple of 4. The address of the heap is returned as the value of this function. The actual size of the heap (in bytes) is returned in `*sizeptr`. The value placed into `*realresultptr` is used as the argument to `freeHeap` when freeing the heap.

  For most ports, `*realresultptr` will be set to the actual value returned by the native space allocation function. If this value is not a multiple of 4, it is rounded up to the next multiple of 4, and `*sizeptr` is decreased by 4.
- `void freeHeap(void *heapPtr)`
  Free the heap space that was allocated using `allocateHeap`. See above for the meaning of the `heapPtr` argument.
- `void GetAndStoreNextKVMEvent(bool_t forever, ulong64 waitUntil)`
  This function serves as an interface between the event handling capabilities of the virtual machine and the host operating system. (Defined in `VmCommon/h/events.h`.) See `Chapter 12` for details.
- `void InitializeVM()`
  Initialize the virtual machine in whatever way is necessary. On many of the current ports, this is a macro that does nothing.
- `void InitializeNativeCode()`
  Initialize the native code in whatever way is necessary. Ports can use this function (for example) to initialize the window system and to perform other native-code specific initialization.

- `void InitializeClassLoading()`
  Initialize the class loader in whatever way is necessary. Ports can use this function (for example) to perform certain file/storage system initialization operations.
- `void InitalizeAsynchronousIO(void)`
  Initialize the system to handle asynchronous native methods.
- `void FinalizeVM()`
  Perform any cleanup necessary before shutting down the virtual machine.
- `void FinalizeNativeCode()`
  Perform any clean up necessary to clean up after the native functions. Many ports use this function to shut down the window system.
- `long RandomNumber_md()`
  Return a random number.
- `void FinalizeClassLoading()`
  Perform any cleanup necessary before shutting the class loader. Ports can use this function (for example) to perform certain file/storage system finalization operations.
- `ulong64 CurrentTime_md(void)`
  Return the time, in milliseconds, since January 1, 1970 UTC. On devices that do not support the concept of time zone, it is acceptable to return the time, in milliseconds, since January 1, 1970 of the current time zone.
- `unsigned long *Calendar_md(void)`
  Initializes the calendar fields, which represent the Calendar related attributes of a date.

The functions `InitializeNativeCode()` and `InitializeVM()` are called, in that order, before any global variables have been set and before the memory-management system has been initialized.

The function `FinalizeVM()` is called just before `FinalizeNativeCode()`. On those ports that have enabled profiling, the profiling information is printed out between the calls to these two functions. This allows the profiler to find out information about the window system, if necessary, and to use the window system for creating its output.

Note – If you want to use the KVM for running additional libraries such as those defined by the *Mobile Information Device Profile (MIDP)* or *PDA Profile*, additional porting work will be necessary to port the native functions required by those libraries.

Asynchronous native functions. If your port supports the use of asynchronous native methods, there are additional, port-specific functions that you must define :

```
yield_md()
CallAsyncNativeFunction_md()
enterSystemCriticalSection()
exitSystemCriticalSection()
```

Note – The interfaces for functions `enterSystemCriticalSection()` and `exitSystemCriticalSection()` are defined in `VmCommon/h/thread.h`.

These functions are further described in §11.4.

4.4 Required C library functions

The KVM uses the following C library functions:

- String manipulation: `strcat`, `strchr`, `strcmp`, `strcpy`, `strncpy`, `strlen`
- Moving memory: `memcpy`, `memove`, `memset`, `memcmp`
- Printing: `atoi`, `sprintf`, `fprintf, putchar`
- Random number generation: `rand`
- Exception handling: `setjmp`, `longjmp` (not absolutely necessary)

If your development environment does not supply definitions for these functions, you must either define them yourself, or use macros to map these names onto equivalent functions recognized by your development environment.[1]

The function `memmove` must be able to handle situations in which the source and destination overlap. The function `memcpy` is used only in those cases in which the source and destination are known not to overlap.

`The functions fprintf and sprintf use the following formats:`

```
%s, %d, %o, %x, %ld, %lo, %lx, %%
```

These formats never have options or flags.

There are no calls directly to `printf`.

Note – The components included in directory `VmExtra` and the machine-specific ports provided with this release may need additional native functions not listed above.

1. Be aware that the order of arguments may be different on different platforms. For example, the function `memset` takes arguments `memset(location, value, count)`. The corresponding Palm OS function is `MemSet(location, count, value)`.

CHAPTER 5

Directory Structure

5.1 Overview

Unzip the release package into any directory of your choice. This will create a directory `j2me_cldc` with the following subdirectories:

- `api`
- `bin`
- `build`
- `doc`
- `jam`
- `kvm`
- `tools`

The contents of these directories are detailed in TABLE 3.

TABLE 3 Distribution directories

Subdirectory	Description
`api`	Contains the Java library source code that is provided with the release.
`bin`	Contains all the binary executables and compiled Java library classes.
`build`	Contains makefiles for building the KVM.
`doc`	Contains documentation.
`jam`	Contains the source code of the optional Java Application Manager (JAM) component that is provided with the KVM.
`kvm`	Contains the source code of the KVM.
`tools`	Contains the source code for a number of tools (JavaCodeCompact, preverifier, KDWP Debug Proxy) that are provided with the release.

5.2 Directory kvm/VmCommon

All common, platform-independent source code of KVM is located in the directory kvm/VmCommon/src/. All common include files are in the directory kvm/VmCommon/h/.

Port specific source and include files should go into the directories kvm/Vm*Port*/src/ and kvm/Vm*Port*/h/, where *Port* is replaced by the name of your platform (e.g., kvm/VmWin, kvm/VmPilot, kvm/VmUnix.)

Some ports may choose to create a kvm/Vm*Port*/build/ subdirectory which holds files that are part of the build process, but are not part of the source code *per se*.

TABLE 4 gives an overview of the KVM source code files contained in kvm/VmCommon/src/ and kvm/VmCommon/h/.

TABLE 4 Files in VmCommon

File	Description
StartJVM.c	Virtual machine startup and command line argument reading.
bytecodes.c	The definition of Java bytecodes for the redesigned bytecode interpreter (since KVM 1.0.2).
cache.h cache.c	Inline caching operations for speeding up method lookup and for supporting “fast” bytecodes.
class.h class.c	Internal runtime data structures and operations for representing Java classes.
events.h events.c	Event system implementation.
execute.h execute.c	Interpreter execution macros and operations needed by the redesigned bytecode interpreter (since KVM 1.0.2).
fields.h fields.c	Internal runtime data structures and operations for representing fields and methods.
fp_math.h fp_math.c	High-level floating point function interface.
frame.h frame.c	Stack frame and exception handling operations.
garbage.h garbage.c collector.c collectorDebug.c	Garbage collector and memory management.
global.h global.c	Miscellaneous global variables and definitions.

TABLE 4 Files in VmCommon

File	Description
hashtable.h hashtable.c	Hashtable implementation that is used internally by the virtual machine.
interpret.h interpret.c	Bytecode interpreter. Note that starting from KVM 1.0.2 the actual interpreter code and bytecode definitions are located in other files (bytecodes.c, execute.h, execute.c).
kni.h kni.c	K Native Interface (KNI) support.
loader.h loader.c	Class loader and class format checks required by the class file verifier.
log.h log.c	Logging / diagnostic operations for debugging and profiling.
long.h	Special macros to handle 64-bit operations in a portable fashion.
main.h	Compilation options and system-wide default settings.
messages.h	Error and warning messages.
native.h native.c nativeCore.c	Native function table operations and core native library functions.
pool.h pool.c	Runtime data structures and operations for representing constant pools.
profiling.h profiling.c	Data declarations and operations for profiling virtual machine execution.
property.h property.c	Operations for accessing Java system properties.
rom.h	Macros needed by the ROMizer (JavaCodeCompact tool).
runtime.h	Function templates for certain machine-specific operations that need to defined for each KVM port.
stackmap.c	Stackmap operations that are used for supporting exact garbage collection.
thread.h thread.c	Internal runtime data structures and operations for multithreading and Java thread management.
verifier.h verifier.c verifierUtil.h verifierUtil.c	Classfile verifier (see Chapter 13 for details).

5.3 Directory kvm/VmExtra

The directory kvm/VmExtra/ contains additional components that are potentially useful to a large number of ports. These files include an implementation of the most commonly needed networking protocols for Windows/Unix, a file interface for supporting class loading on those target platforms that have a regular file system, and a JAR file reader/inflater. This directory also contains the implementation of the Java-level debugger and the KDWP (KVM Debug Wire Protocol) interface.

In addition, the directory defines some optional macros for asynchronous event handling, and defines the virtual machine startup operations that are needed on non-embedded, command line based target platforms such as Windows and Solaris.

A description of the VmExtra files is provided in TABLE 5.

TABLE 5 Files in VmExtra

File	Description
async.h async.c	Macros for supporting asynchronous I/O (see Section 11.4 “Asynchronous native methods” and Section 12.1.4 “Asynchronous notification”).
loaderFile.c	Low-level binding between the file system, class loader and JAR reader for those platforms that have a “real” file system.
main.c	Default main program for those platforms that have a file system and support VM startup from a command line.
jar.h inflate.h inflateint.h inflatetables.h jar.c inflate.c	Jar file reader and inflater (decompressor).
resource.c	Implementation of a stream-based protocol for reading external resources.
debugger.h debugger.c debuggerCommands.h debuggerStreams.h debuggerInputStream.c debuggerOutputStream.c debuggerSocketIO.c	Implementation of the Java-level debugger and the KDWP (KVM Debug Wire Protocol) interface.

5.4 Directory `kvm/VmExtra/src/fp`

The directory `kvm/VmExtra/src/fp` contains library functions that are needed for supporting various floating-point functions required by *CLDC Specification* version 1.1. For more information on floating point support, refer to Chapter 10.

CHAPTER 6

Compilation Flags, Definitions and Macros

This section lists various C preprocessor flags, definitions and macros that are defined in `VmCommon/h/main.h`. Understanding the meaning of these flags helps you in porting efforts, so please read the documentation below and in file `VmCommon/h/main.h`.

Note – Rather than changing the values provided in `VmCommon/h/main.h`, these values should be preferably be overridden in your port-specific `machine_md.h` file.

Also note that in our reference implementation, many of these flags are commonly overridden from makefiles.

For each definition, we give a brief summary and its default definition. These flags and macros are documented also in `VmCommon/h/main.h`.

6.1 General compilation options

The following definitions control the general platform-dependent compiler options that you must set before starting your porting efforts. Incorrect settings typically cause the virtual machine to malfunction.

```
#define COMPILER_SUPPORTS_LONG 1
```

Turn this flag on if your compiler has support for long (64 bit) integers.

```
#define NEED_LONG_ALIGNMENT 0
```

Instructs the KVM to know that your host operating system and compiler generally assume all 64-bit integers to be aligned on eight-byte boundaries.

```
#define NEED_DOUBLE_ALIGNMENT 0
```

Instructs the KVM to know that your host operating system and compiler generally assume all double floating point numbers to be aligned on eight-byte boundaries (this flag is meaningful only if floating point support is turned on.)

Additional notes. The compiler generates better code if it knows the "endianness" of your machine. You should set one of the following two variables to "1" in your machine-specific header file.

```
#define BIG_ENDIAN 0
#define LITTLE_ENDIAN 0
```

It is unnecessary to set one of these "endian" variables to "1" if you reset `COMPILER_SUPPORTS_LONG` to zero. (See Chapter 9 for more details.)

Also note that if your compiler supports 64-bit integer arithmetic and you have set the flag

```
#define COMPILER_SUPPORTS_LONG
```

you should supply definitions for the types `long64` and `ulong64`. If your compiler does not support 64-bit integers (or you have set the flag to `0` for some other reason), structure definitions of these two types are created for you automatically. (See Chapter 9.)

6.2 General system configuration options

The following definitions allow you to control which components and features to include in your port.

```
#define IMPLEMENTS_FLOAT 1
```

Turns floating point support in KVM on or off. Should be '1' in those implementations that are compliant with *CLDC Specification* version 1.1, and '0' in those implementations that are compliant with *CLDC Specification* version 1.0.

```
#define PATH_SEPARATOR ':'
```

Path separator character used in CLASSPATH. This definition is meaningful only when utilizing the default class loader for command line based systems. (Defined in `VmCommon/h/loader.h`.)

```
#define ROMIZING 1
```

Turns class prelinking/preloading (JavaCodeCompact) support on or off. If this option is turned on, KVM prelinks all the system classes directly in the virtual machine, speeding up application startup considerably. Refer to Chapter 14 for details.

`#define USE_JAM 0`

Includes or excludes the optional Java Application Manager (JAM) component in the virtual machine. Refer to Chapter 15 for details.

`#define ASYNCHRONOUS_NATIVE_FUNCTIONS 0`

Instructs the KVM to use optional asynchronous native functions. Refer to Section 11.4 “Asynchronous native methods” and Chapter 12 for details.

`#define USE_KNI 1`

This option was introduced in KVM 1.0.4. When enabled, the system will include some code that is needed by the K Native Interface (KNI). If you do not intend to use KNI (you should!), we recommend you to turn this option off, because old-style native functions will run slightly faster with this option turned off. Refer to the *KNI Specification* for further information on KNI.

6.3 Palm-specific system configuration options

The following definitions allow you to control certain Palm-specific system configuration options. All these features were originally designed for the Palm OS version of KVM, but they may be useful also for other ports.

Note – The CLDC implementation for the Palm OS is no longer available.

`#define USESTATIC 0`

Instructs the KVM to use a Palm-specific optimization in which certain immutable runtime data structures are moved from “dynamic RAM” to “storage RAM” to conserve Java heap space. A fake implementation of this mechanism is available also for the Windows and Solaris versions of KVM (for debugging purposes.)

`#define CHUNKY_HEAP 0`

Instructs the KVM to use an optimization which allows the KVM to allocate the Java heap in multiple chunks or segments. This makes it possible for the virtual machine to allocate more heap space on certain platforms such as Palm OS.

`#define RELOCATABLE_ROM 0`

Instructs the KVM to use an optimization in which the prelinked system classes are stored using a relocatable (movable) representation. This allows romized (JavaCodeCompacted) system classes to be stored in devices such as Palm OS.

6.4 Memory allocation settings

The following definitions affect the amount of memory KVM allocates.

`#define DEFAULTHEAPSIZE 256*1024`

The Java heap size that KVM allocates upon virtual machine startup. This value is commonly overridden from makefiles. Note that, starting from KVM 1.0.3, it is possible to override the heap size value from the command line (in those ports that support command line operation.) The heap size value must be a number that is divisible by four. The number must be in the range of 16k to 64 M.

`#define INLINECACHESIZE 128`

The size of a special inline cache area that KVM reserves upon virtual machine startup if the `ENABLEFASTBYTECODES` option is turned on. The inline caching mechanism speeds up method lookups in the KVM by utilizing a technique popularized by Deutsch & Schiffman in the early 1980s. The size here is expressed as a number of inline cache entries (each entry requires 12-16 bytes depending on your target platform.)

`#define STACKCHUNKSIZE 128`

The execution stacks of Java threads inside the KVM grow and shrink automatically as necessary. This value defines the default size of a new stack frame chunk when a new stack chunk needs to be allocated. Reducing the default stack chunk size will make the creation of new Java threads less expensive, but will slow down the execution of the VM when running programs that require a lot of stack space (that is, programs that have a lot of nested method calls.)

`#define STRINGBUFFERSIZE 512`

The size (in bytes) of a statically allocated area that the virtual machine uses internally in various string operations.

Note – As a general principle, KVM allocates all the memory it needs upon virtual machine startup. At runtime, all the memory is allocated inside the preallocated areas. Of course, the situation may change if the virtual machine calls host-system specific native functions (such as graphics functions) that perform dynamic memory allocation outside the Java heap.

6.5 Garbage collection options

The following option turns on compacting garbage collection. Note that currently compaction cannot be used on those platforms that have a segmented (non-contiguous) memory architecture.

```
#define ENABLE_HEAP_COMPACTION 1
```

The following option, if set to a non-zero value, causes a garbage collection to occur on every allocation. This makes it easier to find garbage collection problems. Since this option makes the virtual machine run extremely slowly, the option should be turned off in production builds.

```
#define EXCESSIVE_GARBAGE_COLLECTION 0
```

6.6 Class loading options

Some KVM ports may want to forbid any new classes from being loaded into any system package. The following macro defines whether a package name is one of these restricted packages. By default, the system prevents dynamic class loading to `java.*` and `javax.*` packages.

```
#define IS_RESTRICTED_PACKAGE_NAME(name) \
((strncmp(name, "java/", 5) == 0) || \
 (strncmp(name, "javax/", 6) == 0))
```

6.7 Interpreter execution options (since KVM 1.0)

The following macros allow you to turn on and off certain features controlling interpreter execution. The default values for a production release are shown below.

```
#define ENABLEFASTBYTECODES 1
```

Turns runtime bytecode replacement and method inline caching on or off. This option improves the performance of the virtual machine by about 10-20%, but increases the size of the virtual machine by a few kilobytes. Note that bytecode replacement cannot be performed on those target platforms in which bytecodes are stored in non-volatile memory such as ROM.

`#define VERIFYCONSTANTPOOLINTEGRITY 1`

Instructs the virtual machine to verify the types of constant pool entries at runtime when performing constant pool lookups. Reduces runtime performance slightly, but is generally recommended to be kept on for safety and security reasons.

Additional definitions and interpreter macros:

`#define BASETIMESLICE`

The value of this variable determines the basic frequency (as a number of bytecodes executed) in which the virtual machine performs thread switching, event notification and other periodically needed operations. A smaller number reduces event handling and thread switching latency, but causes the interpreter to run more slowly.

`#define DOUBLE_REMAINDER(x, y) fmod(x,y)`

A compiler macro, defined in `interpret.h`, that is used to find the modulus of two floating point numbers.

`#define SLEEP_UNTIL(wakeupTime)`

This macro makes the virtual machine sleep until the current time (as indicated by the return value of the function `CurrentTime_md()`) is greater than or equal to the wakeup time. The default implementation of `SLEEP_UNTIL` is a busy loop. Most ports should usually provide a more efficient implementation for battery conservation reasons. Refer to Section 12.4 “Battery power conservation” for further details.

6.8 Interpreter execution techniques (after KVM 1.0.2)

Since the release 1.0.2, KVM has an interpreter design that gives up to 15-30% better performance than KVM 1.0 without any loss of ANSI C portability. The actual performance improvement percentage depends on the target platform and the capabilities of the C compiler that is used for compiling the KVM. The performance improvement is the result of the following four techniques that can be used independently of each other:

- Restructuring the interpreter code so that virtual machine registers will be placed into local C variables when the interpreter is running.
- Splitting uncommonly used Java bytecodes into a separate interpreter loop subroutine. This allows the C compiler to do a better job in optimizing the code for more frequently used bytecodes.
- Moving the test for Java thread rescheduling from the top of the interpreter loop to branch bytecodes. This reduces the overhead of the timeslice counter that is used for controlling thread switching.

- Padding out the bytecode space in order to allow the C compiler to produce better code for the main switch statement of the interpreter.

These techniques do not depend on any compiler-specific features, and are therefore portable across a wide variety of C compilers. Each of the techniques and the corresponding macros are discussed in more detail below.

6.8.1 Copying the virtual machine registers to local variables

The virtual machine registers of the KVM (`ip`, `sp`, `lp`, `fp`, `cp`) are accessed very frequently when bytecodes are being executed. In KVM 1.0, all these virtual machine registers are defined as global C variables. Starting from KVM 1.0.2, these registers are still principally defined as global variables, but if the `LOCALVMREGISTERS` option is on, they are copied to local variables when the interpreter is executing. A good C compiler will then optimize the interpreter loop so that these local variables are put into machine registers for substantially faster execution.

```
#define LOCALVMREGISTERS 1
```

Turns the localization of virtual machine registers on or off.

```
#define IPISLOCAL 1
#define SPISLOCAL 1
#define LPISLOCAL 0
#define FPISLOCAL 0
#define CPISLOCAL 0
```

These macros allow you to control specifically which of the virtual machine registers should be used locally by the interpreter loop. These macros have been added to provide better control over register allocation, as many resource-constrained platforms may not have many physical hardware registers available.

The optimal selection of these options for a specific platform will require careful examination of the machine code produced by the compiler, along with a good deal of experimentation. By default, `ip` (instruction pointer), and `sp` (stack pointer) are allocated locally, while `lp` (locals pointer), `fp` (frame pointer) and `cp` (constant pool pointer) are kept in global variables.

Note – If you use the LOCALVMREGISTERS option and you want to make further changes to the code implementing Java bytecodes, the single most important thing to remember is to make sure that the local copies of the virtual machine registers are copied back to their global variables before calling functions in the virtual machine that expects them to be in their global variables. Failure to do so will lead to obscure bugs. The virtual machine registers can be saved to their global variables by using the macro VMSAVE. They are restored back to their local variables by using the macro VMRESTORE. For instance the RETURN bytecodes may need to call monitorExit(), and to do this the call must be done as follows:

```
VMSAVE
result = monitorExit(...);
VMRESTORE
```

6.8.2 Splitting uncommon bytecodes into a separate subroutine

The KVM 1.0 interpreter had the code for all the Java bytecodes in a single large switch statement. However, a majority of Java bytecodes are executed very rarely. If the code for the more frequently and less frequently used bytecodes is placed in separate routines, the C compiler can often do a better job optimizing the resulting smaller interpreter loops. This also helps the compiler find hardware registers for the virtual machine registers more easily when the LOCALVMREGISTERS option is in use.

```
#define SPLITINFREQUENTBYTECODES 1
```

Turning this option on allows the C compiler to generate separate interpreter loops for the frequently and infrequently used bytecodes.

Note that the code to process the bytecodes is now contained in a file called bytecodes.c. The code for all the bytecodes is kept here and is selectively compiled by utilizing a number of internal macro definitions (STANDARDBYTECODES, INFREQUENTSTANDARDBYTECODES, FLOATBYTECODES and FASTBYTECODES).

The code in bytecodes.c is executed from another new file called execute.c. If the SPLITINFREQUENTBYTECODES option is enabled, the file bytecodes.c is included twice into execute.c: once for the routine called SlowInterpret() and once for the routine Interpret(). The four macros mentioned above are used to control the expansion of the appropriate bytecodes into the correct subroutines.

6.8.3 Moving the test for thread rescheduling to branchpoints

The old KVM 1.0 interpreter tested for the need to reschedule (switch threads) before the execution of each bytecode. The performance of the interpreter was improved by about 5% by changing the location of this test so that the test is performed only after every branch, goto, call and return instruction.

Thread scheduling in the old interpreter took place when a certain number of bytecodes had been executed. This number was, by default, 100 times the priority of the thread. In the new interpreter, thread rescheduling occurs by default when 1000 times the number of branch, call, or return bytecodes have been executed.

```
#define RESCHEDULEATBRANCH 1
```

Turning this option on changes the thread switching mechanism so that tests for thread switching are moved to branchpoints. Note that enabling this option affects the value of the `BASETIMESLICE` macro inherited from KVM 1.0. When this option is off, thread scheduling operates as in KVM 1.0.

6.8.4 Padding out the bytecode space

The Java Virtual Machine Specification defines 200 standard bytecodes, plus additionally reserves four other bytecodes for other use. However, many C compilers produce better code when the size of the bytecode (switch) table is exactly 256.

```
#define PADTABLE 0
```

Turning this option on will pad the interpreter switch tables so that the number of instructions is 256. This will increase the size of the virtual machine, but allows the interpreter to run faster on some platforms.

6.9 Java-level debugging options

The KVM 1.0.2 release introduced a new Java-level debugger interface that allows the KVM to be plugged into third party Java debugger environments and integrated development environments (IDEs) that supports the JDWP (Java Debug Wire Protocol) protocol. The macros in this subsection are related to the Java-level debugger options.

Note – It is important to notice that there is a fundamental difference between the debugging facilities intended for *Java-level debugging* and *VM-level debugging*.

Java-level debugging facilities are related to the debugging of the Java programs that the KVM executes. *VM-level debugging* facilities are used for debugging the KVM itself at the native (C) code level.

`#define ENABLE_JAVA_DEBUGGER 0`

Includes a large amount of debugger support code that is needed for plugging KVM into a third-party Java debugger or integrated development environment such as Forte or Borland JBuilder.

More information about the Java-level debugger facilities and the KDWP interface is provided in Chapter 16, “Java-Level Debugging Support (KDWP).”

6.10 VM-level debugging and tracing options

KVM provides a large number of debugging and tracing facilities that can be used for inspecting the behavior of the KVM itself at the native (C) code level. These facilities can be extremely helpful during porting efforts.

All the VM-level debugging and tracing options should be turned off in a production release.

6.10.1 Including and excluding debugging code

`#define INCLUDEDEBUGCODE 0`

Includes a large amount of debugging and logging code that is useful when porting the virtual machine onto a new platform. This option should be turned off in production builds.

`#define ENABLEPROFILING 0`

Turns on or off certain profiling features that allow you to monitor virtual machine execution and get execution statistics. Turning this option on slows down the virtual machine execution speed considerably. This option should be turned off in production builds.

6.10.2 Tracing options

In KVM 1.0, all the tracing options were compilation flags that could be changed only by recompiling the virtual machine. In KVM 1.0.2, all these tracing options were changed into global variables that can be controlled from the command line. This makes it much easier to turn individual tracing options on and off. These global variables (and command line switches) are available only if the virtual machine has been compiled with the `INCLUDEDEBUGCODE` mode turned on.

TABLE 6 Command line tracing options

Option	Description
`-traceallocation`	trace memory allocation
`-tracedebugger`	trace the debugging interface (since KVM 1.0.3)
`-tracegc`	trace garbage collection
`-tracegcverbose`	trace garbage collection, more verbose
`-traceclassloading`	trace class loading
`-traceclassloadingverbose`	trace class loading, more verbose
`-traceverifier`	trace class file verifier
`-tracestackmaps`	trace the behavior of stack maps
`-tracebytecodes`	trace bytecode execution
`-tracemethods`	trace method calls
`-tracemethodsverbose`	trace method calls, more verbose
`-traceframes`	trace stack frames
`-tracestackchunks`	trace the allocation of new stack chunks
`-traceexceptions`	trace exception handling
`-traceevents`	trace the behavior of the event system
`-tracethreading`	trace the behavior of the multithreading system
`-tracemonitors`	trace the behavior of monitor objects
`-tracenetworking`	trace the network access
`-traceall`	activates all the tracing options above simultaneously

If your target platform does not support command line operation, you can control these options directly by changing their default values in file `VmCommon/src/global.c`, or by defining a graphical user interface that sets and resets these options.

Additionally, you can control whether the tracing messages printed out are terse or more verbose by modifying the following option:

```
#define TERSE_MESSAGES 0
```

KVM also contains a stack trace printing facility that can be turned on to help debugging of exceptions and errors in more detail (at the cost of some additional memory footprint). By default, this mode is turned on automatically when the `INCLUDEDEBUGCODE` flag is turned on.

```
#define PRINT_BACKTRACE 0
```

6.11 Error handling macros

Note – The internal error handling macros used by the KVM have been redesigned in KVM 1.1 to support the redesigned class loader.

The interpreter uses the internal error handling macros shown in CODE EXAMPLE 1.

If there is a call to the macro `THROW(error)`, anywhere inside the "normal code," the VM jumps immediately to error handling code. Uses of this macro can be nested, either lexically or dynamically. The `THROW` jumps to the innermost `CATCH` error handling code. (The various `TRY`, `THROW`, and `CATCH` macros are defined in `VmCommon/h/global.h`.)

CODE EXAMPLE 1 Error handling macros

```
TRY {
   normal code
} CATCH (error) {
   error handling code
} END_CATCH
   always continue here
```

By default, this behavior is emulated using `setjmp` and `longjmp`. However, platforms (such as PalmOS) that already provide a similar mechanism should use the native mechanism.

KVM 1.1 also has new macros for controlling the shutdown of the virtual machine. These macros have been illustrated in CODE EXAMPLE 2

CODE EXAMPLE 2 VM shutdown macros

```
VM_START {
   normal VM code
} VM_FINISH (value) {
   code to execute before VM shuts down
} VM_END_FINISH
```

Rather than calling the normal C `exit` function, the proper way to exit from the VM is to call macro `VM_EXIT(value)`. Calling this macro will cause the control of the VM to be immediately transferred to the code that follows the `VM_FINISH(value)` macro. The *`value`* to be passed to this code typically represents the exit code that the VM will return when it shuts down.

6.12 Miscellaneous macros and options

`#define UNUSEDPARAMETER(var)`

Some functions in the reference implementation take arguments that they do not use. Some compilers issue warnings; others do not. For those compilers that do issue warnings, they differ in how you indicate that the non-use of the variable is intentional and that you do not wish to get a warning. This macro should do whatever is necessary to get your compiler to remain quiet.

6.13 Overriding the compilation flags and other options from makefiles

The following parameters are commonly used when using `gnumake` to build the KVM.

`gnumake ROMIZING=false`

Build the KVM with romizing disabled. That is, do not link all the system classes statically into the KVM executable. (The default is to build the KVM with romizing enabled.)

`gnumake DEBUG=true`

Build the KVM with the Java-level debugger and VM-internal debugging code enabled.

`gnumake USE_JAM=true`

Build the KVM with the Java Application Manager (JAM) enabled.

`gnumake GCC=true`

Use GNU C compiler instead of the standard Sun compiler (on Solaris.)

`GCC=true` is the default option when developing on Linux, and this is the setting for compiling on Windows using CygWin tools.

```
gnumake USE_KNI=false
```

Build the KVM without the K Native Interface (KNI) functionality. (The default is to build the KVM with KNI enabled.)

CHAPTER 7

Virtual Machine Startup

Virtual machine startup practices can vary significantly in different KVM ports. By default, KVM supports regular command line based Java virtual machine startup, but the virtual machine can easily be modified for those environments in which command line based startup is not desired.

7.1 Command line startup

This subsection describes the virtual machine startup conventions when launching KVM from a command line.

The file `VmExtra/src/main.c` provides a default implementation of `main()`. The virtual machine is called from the command line as follows:

```
kvm [option]* className [arg]*
```

where each `option` is one of

```
-version
-classpath <list of directories>
-heapsize <heap size parameter>
```

The required `className` argument specifies the class whose method `static main(String argv[])` is to be called. All arguments beyond the class name are uninterpreted strings that are made into a single `String[]` object and passed as the single argument to the `main` method.

The `-classpath` option allows the user to define the directories from which the KVM reads the class files. The parameter `<list of directories>` is a single string in which the directories are separated by the `PATH_SEPARATOR` character. The value of the `PATH_SEPARATOR` character is typically ';' on Windows platforms, and ':' on Unix platforms.

The `-heapsize` option (introduced in KVM 1.0.3) allows the user to manually set the Java heap size that KVM allocates upon virtual machine startup. The heap size can range from 16 kilobytes to 64 megabytes. The heap size can be specified either

in bytes (such as 32768), kilobytes (such as 32k or 32K), or megabytes (such as 1m or 1M). Note that when the heap size is defined in bytes, the KVM automatically rounds up the heap size number to the next number that is divisible by four.

Additionally, if the virtual machine has been compiled with the `INCLUDEDEBUGCODE` mode turned on, the tracing options given in TABLE 6 on page 29 are available.

When the Java-level debugging interface is in use, additional command line options are available to control the debugger. Refer to Chapter 16 for details.

The default implementation of `main(int argc, char **argv)` calls the function `StartJVM()` with an `argv` in which all of the options have been removed and an `argc` that has been decremented appropriately.

7.2 Alternative VM startup strategies

If your implementation does not start the virtual machine from a command line (for example, if you use a graphical environment for application launching), you must arrange your code to call `StartJVM()` with the appropriate arguments.

7.3 Using a JAM (Java Application Manager)

Many KVM ports run on resource-constrained devices which lack many features commonly available in desktop operating systems, such as a command line language, graphical file manager, or even a file system. To facilitate the porting of KVM to such platforms, KVM provides a sample implementation of a facility called JAM (Java Application Manager).

At the compilation level, JAM can be turned on or off by using the flag

```
#define USE_JAM 1
```

When building the KVM using `gnumake`, the following command automatically builds the system with the JAM enabled:

```
gnumake USE_JAM=true
```

If JAM is compiled into the KVM, it must be activated with the `-jam` command line flag.

The JAM implementation assumes that applications are available for downloading as JAR files by using the HTTP protocol. The JAM reads the contents of the JAR file and an associated descriptor file via HTTP, and launches KVM with the main class as a parameter.

Since the JAM serves as an interface between the host operating system and the virtual machine, it can be used, e.g., as a starting point for a device-specific graphical Java application management and launching environment ("microbrowser"), or as a test harness for virtual machine testing. The JAM reference implementation provides a special "`-repeat`" mode that allows the JAM to run a large number of Java applications (e.g., test cases) without having to restart the virtual machine every time.

Refer to Chapter 15, "Java Application Manager (JAM)," for further information on the JAM.

CHAPTER 8

Class Loading, JAR Files, and Inflation

The KVM source code includes an implementation for reading Java class files from regular files / directories, as well as from (compressed) JAR files. Generally speaking, the KVM class loader can be divided into two parts:

1. generic part,
2. port-dependent part.

The generic part, defined in file `VmCommon/src/loader.c` is designed to be independent of the file / storage system of the target device. This part of the class loader does not require any porting efforts. The JAR file reader, defined in files `VmExtra/src/jar.c`, `VmExtra/src/inflate.c`, `VmExtra/h/jar.h`, `VmExtra/h/inflate.h`, `VmExtra/h/inflateint.h`, and `VmExtra/h/inflatetables.h`, is also written in a way that it does not necessitate any porting efforts.

Note – The generic part of the class loader implementation has been redesigned in KVM 1.1 to support error handling in a more generalized, J2SE-compliant fashion.

If you need to provide an alternative method for loading class files, you must define your own port-specific class loading mechanism. The default implementation in `VmExtra/src/loaderFile.c` is intended for those target systems that have a conventional file system. This implementation can be used as a starting point for alternative, platform-specific implementations.

The KVM code to read JAR files can also be used independently of reading class files. Applications that need to make their own use of JAR files can use these functions. In addition, the function that decompresses compressed JAR entries (a process called “inflation”), can also be used to decompress other information. For example, the PNG image format uses the same compression and decompression algorithms.

8.1 Porting the class file loading interface

The structures and functions required by the port-specific class file loading interface have been defined in file `VmCommon/h/loader.h`. If you do not intend to use the default class file loading interface provided in file `VmExtra/src/loaderFile.c`, you must supply your own definitions for the structures and functions listed below.

You must define the C structure `filePointerStruct`. The generic code uses the definitions

```
struct filePointerStruct;
typedef struct filePointerStruct *FILEPOINTER;
```

without knowing anything about the fields of this structure.

You must also define the following functions:

- `void InitializeClassLoading()`
  The code typically initializes the variable `ClassPathTable` and any other variables needed for file loading upon virtual machine startup. Keep in mind that the value in `ClassPathTable` is usually a root for garbage collection, and must either be `NULL` or be an object allocated from the heap.
- The C preprocessor constant `PATH_SEPARATOR` indicates the character that separates directories in the class path. Its default value is `':'`. If you are using Windows or a similar implementation, you will need to change this value to `';'`. (Defined in `VmCommon/h/loader.h`.)
- `void FinalizeClassLoading()`
  This function is the opposite of `initializeClassLoading()`. This function performs the class loader finalization operations that are necessary when the virtual machine shuts down. Actual implementation will vary substantially depending on the target architecture.
- `FILEPOINTER openClassfile(INSTANCE_CLASS clazz)`
  Open the class file pointed to by the `clazz` pointer.
- `void closeClassfile(FILEPOINTER_HANDLE ClassFileH)`
  Close the indicated class file. Close any system resources (such as file handles or database records) associated with the class file.
- `void loadByteNoEOFCheck(FILEPOINTER_HANDLE ClassFileH)`
  Load the next byte if it is a JAR file, or load the next character and return it, or EOF (-1) if end of file was reached.
- `unsigned char loadByte(FILEPOINTER_HANDLE ClassFileH)`
  `unsigned short loadShort(FILEPOINTER_HANDLE ClassFileH)`
  `unsigned long loadCell(FILEPOINTER_HANDLE ClassFileH)`
  Read the next one, two, or four bytes from the class file, and return the result as an unsigned 8-bit, unsigned 16-bit, or unsigned 32-bit value. 16- and 32-bit quantities in Java class files are always in big-endian format.

- `void loadBytes(FILEPOINTER_HANDLE ClassFileH, char *buffer, int len)`
  Load the next `len` bytes from the class file into the indicated buffer.
- `int loadBytesNoEOFCheck(FILEPOINTER_HANDLE ClassFileH, char *buffer, int pos, int length)`
  Load the next `length` bytes from the class file into the indicated buffer, but without checking for EOF.
- `void skipBytes(FILEPOINTER_HANDLE ClassFileH, unsigned long length)`
  Skip the next `length` bytes in the class file.
- `int getBytesAvailable(FILEPOINTER_HANDLE ClassFileH)`
  Get the number of remaining bytes in the class file.

The class file structure returned by `openClassFile` must be an object allocated from the Java heap.

8.2 JAR file reader

CLDC-compliant KVM implementations are required to be able to read class files from compressed JAR files. The location of the JAR file(s) is specified in an implementation-dependent manner.

Functions are provided in `jar.c` for reading entries in a JAR file. If the preprocessor symbol `JAR_FILE_USE_STDIO` is non-zero, then these functions use C standard I/O routines to read the JAR file. If this preprocessor symbol is set to `0`, this indicates that JAR files are in memory.

The JAR file reader uses the inflater, which is discussed in the next section.

8.2.1 Opening a JAR file

Before using a JAR file, you must “open” it using the function

```
bool_t openJARFile(void *nameOrAddress, int length,
                   JAR_INFO entry)
```

The arguments are as follows:

If `JAR_FILE_USE_STDIO` is non-zero, then the first argument is the name of the JAR file and the second argument is ignored.

If `JAR_FILE_USE_STDIO` is zero, then the first argument is a pointer in memory to the beginning of the JAR file, and the second argument is the length, in bytes, of the JAR file.

The third argument is a pointer to a structure of type `struct jarInfoStruct` defined in `jar.h`. This structure is filled with information about the opened JAR file. This function returns `TRUE` if it successfully managed to open the JAR file and parse its directory; it returns `FALSE` otherwise.

8.2.2 Closing a JAR file

If a JAR file has been successfully opened using `openJARFile`, you must close the file when you are done. You must use the function:

```
void closeJARFile(JAR_INFO entry)
```

The argument is a pointer to the same structure that was filled in by `openJARFile`.

8.2.3 Reading a JAR file entry

To read a specific entry in a JAR file, you use the function

```
static void *
loadJARFileEntryInternal(JAR_INFO entry,
                         const unsigned char *centralInfo,
                         long *lengthP, int extraBytes);
```

The `entry` argument is a pointer to the structure filled in by `openJARFile`. The `centralInfo` argument is the null-terminated name of the entry.[1] The `extraBytes` entry indicates that the JAR reader should pad the result with that many extra bytes at the beginning.

If the JAR file reader is successful, it will set the `*lengthP` argument to the length of JAR file entry. This length does *not* include padding inserted because of the `extraBytes` argument. The actual entry (plus padding) is returned as the result of this function.

If the JAR file reader could not find the entry, or if for some reason it was unable to read the entry, this function returns `NULL`.

The result of this function is a heap-allocated object. If this function is called from within the KVM, then you must protect it, if necessary, from garbage collection.

1. Note that Jar files always use ‘/’ as the directory separator character.

8.2.4 Reading multiple JAR file directory

To read the directory of a JAR file and possibly some of its entries, use the function

```
void loadJARFileEntries(JAR_INFO jarFile,
                        JARFileTestFunction testFunction,
                        JARFileRunFunction runFunction,
                        void* info);
```

The `jarFile` argument is a pointer to the structure filled in by `openJARFile`. The `testFunction` and `runFunction` arguments are callback functions whose use is described below. The `info` argument is not used by the jar directory reader, but is passed on an argument to the `testFunction` and `runFunction` callbacks.

The `testFunction` argument is a callback function that is called on each (non-directory) entry in the JAR file. It is called as follows:

```
typedef bool_t
    (*JARFileTestFunction)(const char *name,
     int nameLength,
     int *extraBytes,
     void *info);
```

The `name` and `nameLength` argument specify the name of entry in the JAR file directory. The `name` argument is *not* null terminated. The value `*extraBytes` is initially zero, but you can change it to a different value to indicate that the result needs to be padded with extra bytes at the beginning. The `info` argument is the same as whatever was passed to `loadJARFileEntries`.

If this function returns `TRUE`, it indicates that you want to read this entry. If this function returns `FALSE`, you do not want to read this entry.

For every entry in which `testFunction` returns `TRUE`, the jar file reader reads the data and calls the `runFunction` as follows:

```
typedef void
    (*JARFileRunFunction)(const char *name, int nameLength,
      void *value, long length, void *info);
```

The `name` and `nameLength` arguments are the same as above. The `value` argument gives the result of reading the JAR file entry. The `length` argument is the length of the JAR file entry, not including any padding bytes. The `info` argument is the same as whatever was passed to `loadJARFileEntries`.

If reading the entry is unsuccessful, then the `runFunction` is called with the `value` argument set to `NULL`.

The `value` argument is allocated on the heap so it must be protected, if necessary, from garbage collection.

8.3 Inflation

The inflate function can be used to decompress streams that have been compressed using the so-called deflation algorithm. This is the compression algorithm commonly used in JAR files and in the PNG image format.

The function that inflates JAR file entries can also be used for other purposes. The function is called with the following arguments.

```
typedef  int  (*JarGetByteFunctionType)(void *);

bool_t inflateData(void *compData, JarGetByteFunctionType
               getByte,
               int compLen,
               UNSIGNED_CHAR_HANDLE decompData,
               int decompLen);
```

This function decompresses a stream of `compLen` bytes into a buffer of `decompLen` bytes. Successive bytes of input are obtained by repeatedly calling

```
getByte(inFile)
```

This function will be called up to `compLen + INFLATER_EXTRA_BYTES` times, where `INFLATER_EXTRA_BYTES` is defined in `inflate.h` to be the constant 4. Any values returned beyond the first `compLen` calls to the function are immaterial.

The argument `decompData` must be a pointer to a buffer handle of at least `decompLen` characters. When using this function, the buffer must either not be in the heap, or `decompData` must be registered with the garbage collector so that `decompData` is updated if the buffer is moved.

This function returns `TRUE` if the decompression is successful, and `FALSE` otherwise.

CHAPTER 9

64-bit Support

We do not require your compiler to support 64-bit arithmetic. However, having a 64-bit capable compiler makes porting much easier.

9.1 Setup

If your compiler supports 64-bit integers, you should define the types `long64` and `ulong64` in one of your platform-dependent include files. The meaning of these two types is shown below in Table 7.

TABLE 7 64-bit types

Type	Description
`long64`	A signed 64-bit integer.
ulong64	An unsigned 64-bit integer.

You should consider setting one of the two compiler constants `BIG_ENDIAN` or `LITTLE_ENDIAN` to a non-zero value. This is only required if you are using the Java Code Compactor, but KVM can produce better code if it knows the endianness of your machine.

For example, using the Gnu C compiler or the Solaris C compiler, you would write:

```
typedef long long long64;
typedef unsigned long long ulong64;
```

Using Microsoft Visual C/C++, you would write:

```
typedef __int64 long64;
typedef unsigned __int64 ulong64;
```

If Your compiler does not support 64-bit integers,[1] you must set the preprocessor constant `COMPILER_SUPPORTS_LONG` to zero. You must define exactly one of `BIG_ENDIAN` or `LITTLE_ENDIAN`[2] to have a non-zero value.

1. Or your code must be strictly ANSI C standard.

The types `long64` and `ulong64` are defined to be a structure consisting of two fields, each an unsigned `long` word, named `high` and `low`. The `high` field is first if your machine is big endian; the `low` field is first if your machine is little endian.

You must define the functions shown in Table 8. If your platform supports floating point, you must also define the functions shown in Table 9.

Any of these functions can be implemented as a macro instead.

TABLE 8 Implementing longs

Function or Constant		Java equivalent
`long64`	`ll_mul(long64 a, long64 b);`	`a * b`
`long64`	`ll_div(long64 a, long64 b);`	`a / b`
`long64`	`ll_rem(long64 a, long64 b);`	`a % b`
`long64`	`ll_shl(long64 a, int b);`	`a << b`
`long64`	`ll_shr(long64 a, int b);`	`a >> b`
`long64`	`ll_ushr(long64 a, int b);`	`a >>> b`

TABLE 9 Implementing both longs and floats

Function or Constant		Java equivalent
`long64`	`float2ll(float f);`	`(long)f`
`long64`	`double2ll(double d);`	`(long)d`
`float`	`ll2float(long64 a);`	`(float)a`
`double`	`ll2double(long64 a);`	`(double)a`

9.2 Alignment issues

When an object of Java type `long` or `double` is on the Java stack or in the constant pool, its address will be a multiple of 4.

Some hardware platforms (such as SPARC) require that 64-bit types be aligned so that their address is a multiple of 8.

If your platform requires that 64-bit integers be aligned on 8-byte boundaries, set

```
#define NEED_LONG_ALIGNMENT 1
```

2. See Jonathan Swift, *Gulliver's Travels, Part I: A Voyage to Lilliput*, for more information on the big-endian, little-endian controversy.

If your platform requires double-precision floating point numbers be aligned on 8-byte boundaries, set

```
#define NEED_DOUBLE_ALIGNMENT 1
```

The compiler can generates better code when these values are `0`.

CHAPTER 10

Floating-Point Support

This chapter contains an overview of the IEEE 754 floating-point standard, Java virtual machine floating-point semantics, and the porting effort required for the implementation of floating-point to various processor architectures. Additionally, the implementation of `strictfp` arithmetic operations is detailed.

Version 1.0 of the *CLDC Specification* did not require floating-point arithmetic in compliant implementations. However, the *CLDC Specification* Version 1.1 does require floating-point, and this chapter describes the implications for porting the floating-point implementation for KVM in CLDC 1.1.

10.1 Introduction

The Java programming language and the Java virtual machine support two floating-point types, 32-bit `float` and 64-bit `double`. The numerical results for operations performed on values of these types are defined by the IEEE 754 standard for binary floating-point arithmetic (IEEE Std 754-1985). While many processor architectures also support IEEE 754, there can be complications mapping Java virtual machine floating-point operations to C code or to hardware instructions implementing those operations. Before describing those complications and their solutions, more background on IEEE 754 is necessary.

10.1.1 IEEE 754 floating-point

Floating-point numbers are a subset of the real numbers; the representable finite floating-point numbers have *sign*, *exponent*, and *significand* fields.[1]

The numerical value of a finite floating-point number is

$(-1)^{sign} \cdot 2^{exponent} \cdot significand$

1. In other floating-point systems, the *significand* is called the *mantissa*.

The *sign* field is 0 or 1. The *exponent* field is an integer; the *significand* field is a binary number greater than or equal to zero and less than two. The IEEE 754 standard defines the ranges for the *exponent* and *significand* values for the `float` and `double` formats. The `double` format has more than twice the precision of `float` as well as a greater exponent range. To avoid multiple representations for the same numerical value, a floating-point number's representation is *normalized*; that is, the *exponent* is adjusted to the least value so that the leading bit of the *significand* is 1 instead of 0. The *significand* is less than 1 only for *subnormal* values, which are values so small that an in-range exponent cannot be made small enough to normalize the value's representation.

Since floating-point numbers have a fixed amount of precision, there must be a rounding policy to decide which floating-point number to return when storing the exact result requires more bits than the precision of the floating-point format. For example, multiplying two floating-point values can lead to the exact product having twice as many bits as either input. The IEEE 754 default rounding policy used in the Java virtual machine is to return the floating-point value closest to the exact numerical result. However, not all operations have clear finite results. For example, what is 1/0 or 0/0? For such situations, the IEEE 754 standard has the special values *infinity* and *NaN* (not a number). A signed infinity is returned when the exact result is too big to represent (overflow) or when a finite non-zero value is divided by zero. A NaN is returned for invalid operations, such as 0/0 or sqrt(-1). By adding infinities and NaN, IEEE 754 arithmetic forms a closed system. For every set of inputs, an IEEE 754 arithmetic operation returns an IEEE 754 value.

For two IEEE 754 numbers to be equivalent, they must either be the same non-finite value (+infinity, -infinity, NaN) or if both values are finite, each field of the floating-point numbers must be the same.

10.1.2 Implementing Java virtual machine floating-point semantics

Many processor architectures natively support IEEE 754 arithmetic on `float` and `double` formats. Therefore, there is often a straightforward mapping between Java virtual machine floating-point operations, C code implementing those operations, and floating-point instructions on the underlying processor. However, various complications are possible:

- The floating-point semantics of the Java virtual machine are tightly specified, much more tightly specified than C language floating-point semantics. Therefore, a C compiler could perform an "optimization" that was an allowed transformation in C but broke Java virtual machine semantics. For example, in the Java virtual machine and Java:

  x + 0.0

  cannot be replaced with

  x

since different answers can be generated.[1]

Therefore, the portions of the KVM that implement Java virtual machine floating-point semantics should be compiled without aggressive optimization to help avoid such (in this case) unhelpful code transformations. Many C compilers also have separate flags affecting floating-point code generation, such as flags to improve floating-point consistency and make the generated code have semantics more closely resembling a literal translation of the source. Regardless of the processor architecture, using such flags might be necessary to implement Java virtual machine semantics in C code.

- Certain architectures provide floating-point instructions other than addition, subtraction, multiplication and division on `float` and `double` values. For example, many architectures provide a *fused mac* (fused multiply-accumulate) operation instead of (or perhaps in addition) to the standard arithmetic operations. *Fused mac* is a ternary operation that produces ($a \times b + c$) with a single rounding error (as opposed to two rounding errors if the multiply and add are performed separately). The IA-32 (that is, x86) line of processors has a set of floating-point registers in the x87 FPU (floating-point unit) with both more range and more precision than `double`.

Both fused mac and the extra range of the x87 registers necessitate extra care when implementing Java virtual machine semantics.

10.1.3 Java virtual machine floating-point semantics: `strictfp`

There are actually two flavors of floating-point semantics in the Java virtual machine: *FP-strict* semantics and default semantics. *FP-strict* semantics are used if a method or constructor has the `ACC_STRICT` bit set in the *access_flags* field of the *method_info* structure.[2] In Java, this bit gets set if a class or a method is declared `strictfp`. All the floating-point operands and results in *FP-strict* methods and constructors are exactly 32-bit `float` or 64-bit `double` quantities.

In contrast, in default floating-point semantics, while floating-point variables must hold exactly `float` or `double` values, values on the operand stack are allowed, but not required, to have greater exponent range.

The Java programming language provides the `strictfp` modifier, to be applied to the declaration of a class, interface or method containing variables that might take a floating-point value. If the `strictfp` modifier is used, any compile-time expression involving the variables of the declared class, interface or method is said to be *FP-strict*. To be *FP-strict* means that all intermediate floating-point values must be elements of a `float` value set or a `double` value set, implying that the results of all *FP-strict* expressions must be those predicted by IEEE 754 arithmetic

1. In IEEE 754 floating-point, there is both -0.0 and +0.0. These values can be distinguished by division; 1.0 / -0.0 is *negative* infinity while 1.0 / + 0.0 is *positive* infinity. If x is -0.0, x + 0.0 is +0.0; adding + 0.0 to a number changes negative zero into positive zero while leaving other values unchanged.

2. The `strictfp` Java modifier and the `ACC_STRICT` modifier were added in Java 2. Java classes generated before that time will not have *FP-strict* semantics.

on operands represented using `float` (single-precision) and `double` (double-precision) formats. Within an expression that is **not** *FP-strict*, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results. The net effect, roughly speaking, is that a calculation might produce "the correct answer" in situations where exclusive use of the `float` value set or `double` value set might result in overflow or underflow.

For more details, see the *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999) and the *The Java™ Language Specification* by James Gosling, Bill Joy, and Guy L. Steele (Addison-Wesley, 1996).

10.1.4 Floating-point architectures

10.1.4.1 Fused Mac

In general, a *fused mac* cannot be used to implement chained multiply and add instructions in the Java virtual machine since the rounding behavior will be different. This is true for both default and *FP-strict* semantics. However, even if an architecture only has *fused mac* instructions for floating-point, implementing the semantics of separate add and multiply is fairly direct. The result of *(a + c)* is the same as *(a × 1.0 + c)*. The result of *(a × b)* is *almost* the same as *(a × b + 0.0)*; it will be different if *(a × b)* results in a negative zero. Adding a positive zero would result in a positive zero instead of negative zero being returned for the logical product. This discrepancy is not allowed by Java virtual machine semantics. Assuming the "round to nearest" rounding mode is in effect, *(a × b - 0.0)* gives the same result as *(a × b)* even if *(a × b)* is zero. More generally, *fused mac*-based architectures usually have some special instruction idiom to avoid this discrepancy regardless of rounding mode. C compilers for fused mac platforms usually include a switch to disable the collapsing of chained multiplies and adds into fused macs.

10.1.4.2 x87 FPU

The floating-point load and store instructions on the x87 support three floating-point formats 32-bit `float` (8-bit exponent), 64-bit `double` (11-bit exponent), and 80-bit `double extended` (15 bit exponent). However, when values are loaded in the 80-bit registers, they always have 15-bit exponents even when the FPU is set to round to `float` or `double` precision. When implementing Java virtual machine instructions, the x87 FPU should be set to round to `float` or `double` precision. However, especially in *FP-strict* methods, the effect of the additional exponent bits must be compensated for.

10.1.4.2.1 FP-strict

FP-strict instructions must generate the same results everywhere, including x87 FPUs. The extra exponent range complicates this since the overflow threshold (the point at which infinity is returned) and the underflow threshold (the point at which subnormal results are returned) differ with the larger exponent range. For example, if the extra exponent range were not an issue, the `double` computation $d = a \times b + c$ might get translated into a sequence of x87 instructions like

```
# Sample code
fld a  # load a onto register stack
fmul b # multiply a×b and put result on register stack
fadd c # add c to product of a and b and put result on register stack
fst d  # store a×b+c from register stack into d
```

The problem with this code sequence is that the intermediate values $a \times b$ and $(a \times b) + c$ will not overflow or underflow the same way as pure `double` code since the intermediate values are kept in registers with larger exponent range. The first attempt at a solution stores each intermediate product to a `double` location in memory:

```
# Attempted Fix 1
fld a  # load a onto register stack
fmul b # multiply a×b and put result on register stack
fst t1 # store a×b into a temp double location to restrict exponent
fld t1 # reload a×b with restricted exponent
fadd c # add c to product of a and b and put result on register stack
fst d  # store a×b+c from register stack into d
```

This first attempted fix does preserve the proper overflow behavior for $a \times b$. However, the underflow behavior is slightly wrong. Performing the multiply and rounding, storing to restrict the exponent (thus rounding again), and then reloading the stored value can give a different subnormal number than if the product were rounded only once to the final precision and range. The *compute-store-reload idiom* works for addition and subtraction. However, multiplication and division both share this double-rounding-on-underflow hazard. Avoiding the hazard requires a few additional steps; however expressing the needed steps in a C program may be difficult.

If the operand values are `float` instead of `double`, and if the FPU's rounding precision is set to `double` precision, and the loads and stores are of `float` values, the *store-reload idiom* works for the four basic `float` arithmetic operations (*add*, *subtract*, *multiply* and *divide*). In the case of *multiply*, a `double` precision product of `float` operands is exact, so double-rounding is avoided. In general, `double` has enough additional precision over `float` that these double-rounding problems are all avoided.

To avoid double-rounding on underflow for `double` values, what would be a subnormal result in pure `double` must also be a subnormal in the register format with extended exponent range. This can be arranged by scaling one of the operands by a power of two.

```
# Attempted Fix 2
fld a           # load a onto register stack
fmul SCALE_DOWN # scale a down
fmul b          # multiply a_scaled×b, put result on register stack
                #  significand will have the right bits if a×b
                #  should be subnormal
fmul SCALE_UP   # rescale product to restore the proper exponent
fst t1          # store a×b into a temporary double location to
                #  restrict exponent
fld t1          # reload a×b with restricted exponent
fadd c          # add c to product of a and b
                #  and put result on register stack
fst d           # store a×b+c from register stack into d
```

Multiplying by `SCALE_DOWN` and `SCALE_UP` ensures the right result when the product in pure `double` would be a subnormal. The store and reload to and from `t1` is still needed to ensure an overflow to infinity occurs at the proper value.

The magnitude of the exponent of `SCALE_DOWN` and `SCALE_UP` is the difference in the maximum exponent of the `double` format and the maximum exponent of the register format:

`SCALE_DOWN` = $2^{-(E_{max\ register} - E_{max\ double})} = 2^{-(16383 - 1023)} = 2^{-15360}$

`SCALE_UP` = $2^{(E_{max\ register} - E_{max\ double})} = 2^{(16383 - 1023)} = 2^{15360}$

Unfortunately, these values are too large to represent as `double` values. However, they can be easily synthesized out of `double` values if the intermediate products are kept on the FPU stack with its large exponent range:

$2^{-15360} = (2^{-960})^{16}) = ((((2^{-960})^2)^2)^2)^2$

$2^{15360} = (2^{960})^{16}) = ((((2^{960})^2)^2)^2)^2$

2^{-960} = 1.0261342003245941E-289 = `longBitsToDouble`(0x3f0000000000000)

2^{960} = 9.745314011399999E288 = `longBitsToDouble`(0x7bf0000000000000)

As 80-bit values, logically the final bit patterns from most to least significant bit, are:

2^{15360} = 0x7bff 8000 0000 0000 0000

2^{-15360} = 0x03ff 8000 0000 0000 0000

Adjusting by the scaling factors is also needed to implement *divide*. The product or quotient must first be scaled down. Scaling up first will not preserve the underflow threshold.

10.1.4.2.2 Generating FP-strict code in C

If a Java virtual machine on the x87 is generating assembly or machine code directly, creating the code necessary for *FP-strict* semantics is straightforward. However, coaxing the needed instructions from C source can be challenging due to numerous factors:

- Some C compilers do not support a long double type corresponding to the 80-bit (`double extended`) format; they only support `float` and `double`. Therefore, the `SCALE_DOWN` and `SCALE_UP` factors must be created from `double` values.
- Although the scaling factors can be created from `double` values, the computations creating the scaling factors must occur on the x87 stack; any intermediate store to `double` would generate an infinity.
- The above two points imply that the scaling factors (logical constants) might have to be created at runtime. Therefore, the C compiler's optimizer must be turned off to avoid unwanted constant folding.
- If the scaling factors are generated at runtime from `double` values, there is no guaranteed idiom to keep the intermediate values on the stack and to refer to other stack-only values, making using the successive squaring formula problematic.

One approach to dealing with these issues is to generate the scaling factors by multiplying together sixteen copies of $2^{\pm 960}$ stored as a volatile variable. Declaring a variable volatile forces it to be reread every time it is used, foiling unwanted optimizations. However, this means that an *FP-strict* multiply or divide would require (32 + 2) multiplies in addition to the operation being implemented. If `asm` cannot be used to implement the *FP-strict* multiply and divide operations, it may be faster to use an integer-based software implementation of those operations.

10.1.4.2.3 Default floating-point

Compared to *FP-strict* code, generating code with default floating-point semantics is simple. For default code, the scaling factors are not required and the stores and reloads are only necessary for variables. In other words, the stores and reloads are not necessary for quantities that live on the Java virtual machine operand stack.

10.1.4.3 Other architectures

On architectures with only plain `float` and `double` arithmetic operations, mapping to Java virtual machine semantics to equivalent C code is not complicated.

10.2 Floating-point support in the virtual machine

For CLDC 1.1 compliant implementations, the floating-point functionality is always enabled by default. It can be disabled by changing the `IMPLEMENTS_FLOAT` flag in `main.h`. The majority of the support needed in the virtual machine for implementing floating-point is done to the Java bytecodes defined in `bytecodes.c`. The specific modifications needed are described in the sections below.

10.2.1 Floating-point bytecodes implementation

The file `bytecodes.c` represents one of the major components that must be changed to support floating-point. This file contains Java bytecodes executed by the KVM interpreter. Many of the modifications involve checking for NaNs. Among the bytecodes that require modifications are *D2I*, *D2L*, *F2I*, and *F2L*. The modifications and checks for NaNs are described in Section 10.4 "Porting." The x86 specific changes are implemented in `fp_bytecodes.c` (located in directory `kvm/VmExtra/src/fp`). Specific details of the changes are also documented with comments in that file.

10.3 CLDC 1.1 floating-point libraries and trigonometric functions

This section describes the floating-point libraries and the trigonometric and other math functions that are now supported by KVM. The Java classes that are needed for floating-point support are described in the following table:

TABLE 10 Java classes needed for floating-point

File	Description
`Float.java`	Supports floating-point arithmetic.
`Double.java`	Supports double-precision floating-point arithmetic.
`Math.java`	Additional trigonometric and other math functions.
`FloatingDecimal.java`	Used to convert decimals and doubles to strings

These files are not implementation-specific.

The table below lists the trigonometric function that are now implemented in the KVM for floating-point support. Listed with each function are the corresponding file(s) in which the function is implemented.

TABLE 11 Files implementing trigonometric and other math functions

Function	File(s)
sin	k_sin.c s_sin.c
cos	k_cos.c s_cos.c
tan	k_tan.c s_tan.c
sqrt	e_sqrt.c w_sqrt.c
ceil	s_ceil.c
floor	s_floor.c
abs	s_fabs.c

The implementation of the trigonometric functions is taken directly from the JDK1.3.1 sources with no changes except to the function names. The trigonometric files are specified in directory `kvm/VmExtra/src/fp`.

Note – You *cannot* use optimization when compiling the floating-point files. Doing so will cause incorrect results in the trigonometric functions. This means you cannot set `FP_OPTIMIZATION_FLAG`. Refer to Section 10.1.2 “Implementing Java virtual machine floating-point semantics” for further details.

10.4 Porting

The following sections summarize the porting effort required for the implementation of floating-point to various processor architectures. The biggest challenge in the porting effort is in the implementation for handling NaNs and infinity bounds checking. The key changes that are required on all platforms are essentially in the conversion of bytecodes *D2I, D2L, F2I,* and *F2L*. These bytecodes needed additional checks mandated by the *Java™ Virtual Machine Specification* to check for NaNs and infinity bounds, and to return the correct value for each of these cases.

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999) states that for each of these conversion bytecodes, if a NaN value is being converted, the result of conversion is zero. If a value is of large magnitude or small magnitude (such as positive or negative infinity) the maximum or minimum value of the conversion type is the result. In all

other cases, the value is converted from one type to the other using the IEEE 754 conversion rules. The values defined as NaN and infinity are described in the *Java™ Virtual Machine Specification*, §4.4.4 and §4.4.5.

10.4.1 SPARC

The SPARC architecture is IEEE 754 compliant and has direct support for `float` and `double` operations. Therefore, implementing floating-point on KVM/SPARC only requires additional checks for NaN and infinity in the conversion bytecodes, D2I, D2L, F2I, and F2L.

10.4.2 ARM

The ARM CPU uses a IEEE 754 compliant software floating-point library. Similar to SPARC architecture, the only required changes are additional checks to the floating-point conversion bytecodes *D2I*, *D2L*, *F2I*, and *F2L*.

10.4.3 x86

The traditional x87 FPU is fully IEEE 754 compliant. However, the IEEE 754 standard explicitly allows rounding to reduced precision, but greater exponent range, which does not always match the floating-point model used in the Java language and the JVM. Therefore, additional work is needed to implement floating-point. Additionally, the P4 processor contains the SSE2 instruction set extension, which is another IEEE 754 compliant implementation. However, SSE2 is more amenable to Java's semantics.

To implement floating-point for the x86 platform, checks involving NaNs are needed for the following Java bytecodes: *FCMPL*, *FCMPG*, *DCMPL*, *DCMPG*, *FREM*, and *DREM*. These bytecodes needed additional checks to behave as mentioned in the *Java™ Virtual Machine Specification*. The *Java™ Virtual Machine Specification* describes what each of these bytecodes should do or return when a NaN value is encountered.

The file `fp_bytecodes.c` under `kvm/VmExtra/src/fp` contains the x86-specific implementation for the floating-point bytecodes. Each function in this file implements an algorithm for a specific floating-point bytecode that needs modification. Each of these bytecodes check the value that is on the stack to see if it is a NaN. If a NaN value is encountered, it is handled as a special case according to the *Java™ Virtual Machine Specification*. These functions are executed only if the variable `PROCESSOR_ARCHITECTURE_X86` is set in the platform-specific header file `machine_md.h`.

10.4.3.1 *strictfp* implementation for x86

Due to the reasons mentioned in the above sections, the implementation of `strictfp` is quite a challenge for the x86 platform. The x86 is designed to operate on 80-bit `double extended` floating-point values rather than the 64-bit and 32-bit `double` and `float` values used in the Java programming language. The x86 can be made to round to `float` or `double` precision. Unfortunately, this rounding does not exactly emulate the pure `float` and `double` called for by Java, since an extended exponent range is available. The extended exponent range means the overflow and underflow thresholds are different than for pure `float` and `double`.

To implement `strictfp`, the bytecodes *DMUL* and *DDIV* must be changed. The problem is, while doing these operations on subnormal numbers (very small IEEE 754 values with less precision than normal numbers) rounding occurs, producing an incorrect result. (Refer to 10.1.4.2.1, "*FP-strict*.") In addition, double-rounding can occur if the obvious code generation algorithm is used. The solution is to implement the following algorithms for *DMUL* and *DDIV*.

Multiply (DMUL)

- load multiplier
- scale multiplier by multiplying multiplier by 2^{-15360}
- load multiplicand
- multiply scaled multiplier by multiplicand
- rescale product by 2^{15360}
- store rescaled product
- reload stored rescaled product

Divide (DDIV)

For strictfp floating-point on x86, the initial scaled quotient must be smaller than the actual quotient for the rounding to work properly. Thus, the algorithm is:

- load dividend
- load divisor
- compute initial_quotient by either:
  - initial_quotient = $(2^{-15360} \times \text{dividend}) / \text{divisor}$
  - initial_quotient = $\text{dividend} / (\text{divisor} \times 2^{15360})$
- rescale initial_quotient to get the right significand bits. Compute:

  $$\text{quotient} = \text{initial\_quotient} \times 2^{15360}$$

- store rescaled quotient
- reload stored rescaled quotient

The bytecodes for *FADD* and *FSUB* did not need to be changed since if those operations have subnormal results, the results are exact (that is, no rounding occurs).

CHAPTER 11

Native Code

A Java virtual machine commonly needs access to various native functions in order to interact with the outside world. For instance, all the low-level graphics functions, file access functions, networking functions, or other similar routines that depend on the underlying operating system services typically need to be written in native code.

The way these native functions are made available to the Java virtual machine can vary from one virtual machine implementation to another. In order to minimize the work that is needed when porting the native functions, the *Java Native Interface*[1] (JNI) standard was created. The Java Native Interface generally serves two purposes: 1) JNI serves as a common interface for virtual machine implementers so that the same native functions will work unmodified with different virtual machines; 2) JNI provides Java-level APIs that make it possible for a Java programmer to dynamically load libraries and access native functions in those libraries.

Unfortunately, because of its general nature, JNI is rather expensive and introduces a significant memory and performance overhead to native function calls. Also, the ability to dynamically load and call arbitrary native functions from Java programs could pose security problems in the absence of the full Java 2 security model.

KVM does not support the Java Native Interface (JNI). Rather, KVM supports an interface called *K Native Interface* (KNI), which implements a logical subset of JNI that is significantly more efficient in terms of performance and memory consumption. In addition, KVM also supports an older interface that allows native functions to be added to the KVM in a VM-specific fashion. Information on KNI is provided below in Section 11.1 “Using the K Native Interface (KNI)”. Information for writing native functions using the old-style native interface is provided in Section 11.2 “Implementing old-style native methods.”

1. *The Java Native Interface: Programmer’s Guide and Specification (Java Series)* by Sheng Liang (Addison Wesley, 1999).

11.1 Using the K Native Interface (KNI)

Starting from KVM 1.0.4, KVM has a new interface for writing native functions. The high-level goal of this new interface, *K Native Interface (KNI)*, is to allow native functions to be added to the KVM (and other small-footprint virtual machines) in a manner that is both highly efficient and fully independent of the internal structures of the virtual machine. The *K Native Interface (KNI) Specification*, (Sun Microsystems, Inc., 2001) (*KNI Specification)* defines a logical subset of the Java Native Interface (JNI) that is well-suited for low-power, memory-constrained devices. KNI follows the function naming conventions and other aspects of the JNI as far as this is possible and reasonable within the strict memory limits of CLDC target devices and in the absence of the full Java 2 security model. Since KNI is intended to be significantly more lightweight than JNI, some aspects of the interface, such as the parameter passing conventions, have been completely redesigned and are significantly different from JNI.

The K Native Interface is described in more detail in the *KNI Specification*. Please refer to this specification to learn about the K Native Interface and its usage.

Things to remember when getting started with KNI: One of the key goals of the KNI is to isolate the native function programmer from the implementation details of the virtual machine. Instead of writing native functions using KVM-specific functions and data structures (as required by the old-style native interface described in Section 11.2 “Implementing old-style native methods”), KNI allows native functions to be written using a set of functions that operate identically and efficiently across a wide variety of virtual machines. To ensure portability of native code, the native function programmer *shall not use any KVM-specific include files or KVM-specific functions or data types*. Rather, the programmer must include the file “`kni.h`” and use functions and data types defined in that file.

11.2 Implementing old-style native methods

Note – It is highly recommended that you use KNI for writing native functions to the KVM. The use of the old-style API is strongly discouraged for all other purposes than for writing asynchronous native functions (Section 11.4 “Asynchronous native methods”).

WARNING: You should not write old-style native methods unless you have thoroughly read through the implementation and understand its structures. Most of the material in this porting guide is moderately straightforward. The material in this subsection is not!

Old-style native methods must be written extremely carefully. Inattention to detail will cause fatal errors in the virtual machine.

11.2.1 Include files

Your code containing old-style native functions should begin with the line

```
#include <global.h>
```

which causes all include files that are part of KVM to be included. You might also need to `#include` additional files.

11.2.2 Accessing arguments from old-style native methods

When a native method is called, its arguments are on top of the Java stack. A static method's arguments should be popped from the stack in the *reverse order* from which they were pushed. CODE EXAMPLE 3 shows an example of this coding style:

CODE EXAMPLE 3 Handling arguments of native static methods

Java code:

```
static native void
drawRectangle(int x, int y, int width, int height);
```

Native implementation:

```
static void Java_com_sun_kjava_Graphics_drawRectangle() {
    int height = popStack();
    int width =  popStack();
    int y =      popStack();
    int x =      popStack();
    windowSystemDrawRectangle(x, y, width, height);
}
```

An instance method (non-static method) must pop the `this` argument off the stack after it has popped the rest of the arguments.

Note – Failing to pop the `this` argument in a native instance method will almost surely cause a fatal error in the virtual machine.

Table 12 shows the macros that should be used to pop arguments off the stack:

TABLE 12 Macros for popping arguments from the stack

C type	Macro for popping
`char`, `byte`, `int`, `long`	`popStack()`
`float`	`popStackAsType(float)`
`long64`, `ulong64`	`popLong()`
`double`	`popDouble()`
`pointerType`	`popStackAsType(pointerType)`

11.2.3 Returning a result from an old-style native function

If a native method returns a result, it must push that result onto the stack. The native code should use the appropriate macro shown in Table 13 to push the result back onto the stack:

TABLE 13 Macros for pushing arguments onto the stack

C type	Macro for pushing
`char`, `byte`, `int`, `long`	`pushStack()`
`float`	`pushStackAsType(float)`
`long64, ulong64`	`pushLong()`
`double`	`pushDouble()`
`pointerType`	`pushStackAsType(pointerType)`

11.2.4 Shortcuts

Some native code uses the macro `topStack` instead of popping the last argument off the stack. It then sets `topStack` to the value it wants to return.

This practice is not encouraged. It should only be used for “one-liners” that access the argument and return the value in a single statement. `pushStack` and `popStack` cannot be used in this case, since C would not guarantee their order of evaluation.

In general, it is safer to pop the value, perform the calculation, and push the value back onto the stack as three separate steps.

11.2.5 Callbacks

Native code cannot call back into Java code. KVM provides a mechanism by which native code can alter the interpreter state to begin executing a new piece of code. Upon finishing executing that code, the mechanism can indicate a new C function which should be called.

11.2.6 Exception handling in old-style native code

If the native code needs to throw an error or exception, it should call the function

```
void raiseException(const char* exceptionClassName)
```

where the `exceptionClassName` argument is the `exception` class or `error` class.

11.2.7 Useful functions in old-style native code

Other useful functions that a native method might need to call are the following:

- `void fatalError(const char* errorMessage);`
  The code calls this method to indicate that a serious error has occurred. The `errorMessage` argument is a brief explanation of the problem. This method does not return.
- `CLASS getClass(const char *name);`
  This method returns the class whose name is the indicated argument. You might want to coerce the return result to be an `INSTANCE_CLASS` or an `ARRAY_CLASS`.
- `STRING_INSTANCE instantiateString(const char* string, int length);`
  This method converts the given C string into a Java `string`.
- `char *getStringContents(STRING_INSTANCE string);`
  The instance argument must be a Java `string`. It is converted into a null-terminated C string, and returned as the result.

  The string is placed into a global buffer. If your code must hold onto this string for any length of time, you must copy the buffer into stack-allocated storage, or allocate space from the Java heap.
- `INSTANCE instantiate(INSTANCE_CLASS class);`
  Creates a new Java instance of the specified class.
- `ARRAY instantiateArray(ARRAY_CLASS arrayClass, long length);`
  Creates a Java array of the specified type and length.

- `SHORTARRAY createCharArray(const char* utf8stringArg, int utf8length, int* unicodelengthP, bool_t is_permanent);`
  Creates a Java character array from the C string passed as an argument.
- `char* mallocBytes(long sizeInBytes);`
  Allocates a memory block in the garbage-collected heap that is big enough to hold `sizeInBytes` number of bytes. You can create a temporary root (Section 11.2.8 “Garbage collection issues”) to prevent the memory block from being garbage-collected.

11.2.8 Garbage collection issues

The C stack is not scanned when the KVM performs a garbage collection. If your native code allocates new Java objects, you must take special precautions to prevent your new Java objects from being garbage collected inadvertently.

Since the release 1.0.2, KVM includes a compacting garbage collector. Any time that your native code performs an allocation, objects in the Java heap can move. This includes any arguments passed to your native function and any previous heap allocations performed by your native code.

Note – We strongly recommend that you do not write native methods that perform allocation from the Java heap. You greatly increase the chances that your code will have hard-to-find and hard-to-reproduce bugs.

Note – If, for example, you need to create a structure, it is better to create that structure in Java code, and pass it as an argument to the native code.

If your code must perform allocation, it is important that you

- Pop all arguments off the stack before you perform any allocation.
- Push the return value (if any) onto the stack after you have performed any allocation.

The garbage collector can get erroneous results if an allocation occurs while an argument or return value is on the Java stack. The rest of this chapter describes how your code can interact correctly with the garbage collector.

11.2.8.1 Heap Space and Permanent Space

In order to simplify the garbage collector, the KVM’s memory is divided into two spaces: “permanent space” and “heap space”.

All objects created in permanent space are, well, permanent. These objects are

- never freed by the garbage collector,
- never scanned by the garbage collector to see if they contain pointers to other objects,
- never relocated.

Among the objects that are allocated in permanent space are

- class structures,
- Java byte codes,
- method tables,
- field tables,
- interned instances of `java.lang.String` (but not all strings).

These objects are never moved and never freed after they are created.

Structures that have a possibly limited lifetime are allocated in heap space. Among these are

- all Java instances (except for interned `String`s),
- threads,
- stack chunks.

These structures are liable to move any time an allocation occurs. Your code must be following the rules specified in the following subsections to ensure that your code lives happily with the garbage collector.

11.2.8.2 Asserting no allocation

The KVM provides the two macros `ASSERTING_NO_ALLOCATION` and `END_ASSERTING_NO_ALLOCATION`, which are used as shown in CODE EXAMPLE 4:

CODE EXAMPLE 4 Forbidding garbage collection

```
ASSERTING_NO_ALLOCATION
    non allocating code
END_ASSERTING_NO_ALLOCATION;
```

These macros are provided for use only in `DEBUG` mode to guarantee that no allocation is performed by the code between the `ASSERTING`... and the `END_ASSERTING`... macro.

If your code is compiled with `INCLUDEDEBUGCODE` set to a non-zero value, then any allocation inside the specified code causes a fatal error.

If you use the macros, make sure that the non-allocating code inside the macros does not perform a `return`. The macro `END_ASSERTING_NO_ALLOCATION` contains cleanup code that must be executed.

You are encouraged to use these macros to indicate safe regions of code in which heap-allocated objects will not move.

11.2.8.3 Handles in old-style native functions

To deal with the fact that heap-allocated objects in the KVM can move, the garbage collector makes use of temporary "handles." A handle is an indirect pointer to an object. Rather than being the address of the object itself, a handle is the address of a memory location that contains the address of the object.

The memory location that contains the address of the object must not itself be in the Java heap. In general, it is the address of a variable (for global roots) or the address of a location on the C stack (for temporary roots).

- If the object is possibly in the Java heap, then the memory location that contains the address of the object must be registered with the garbage collector. It can either be a temporary root (see §11.2.8.4) or a permanent root (see §11.2.8.5).
- If the object is not in the Java heap, then the handle does not need to be registered with the garbage collector.

All type names in the KVM that end with `_HANDLE` indicate handles. If an argument has a handle as one of its arguments, the argument must be an indirect pointer, and must be registered with the garbage collector if the object could be in the Java heap.

CODE EXAMPLE 5 shows an example:

CODE EXAMPLE 5 Creating a handle

```
CLASS getClassX(CHAR_HANDLE name, int start, int length);

/* Case 1, We are calling it with an argument that is known */
/*         not to be in the heap. */
const char *x = "java/lang/Object";
result = getClassX(&x, 0, strlen(x));

/* Case 2. We are calling it with a heap argument */
START_TEMPORARY_ROOTS
    DECLARE_TEMPORARY_ROOT(char *, x, mallocBytes(100));
    sprintf(x, "java/lang/%s", arg);
    result = getClassX(&x, 0, strlen(x));
END_TEMPORARY_ROOTS
```

11.2.8.4 Temporary Roots

The most common method is to use `START_TEMPORARY_ROOTS` and `END_TEMPORARY_ROOTS` to delimit a region of code. Within this region of code, the macro

```
DECLARE_TEMPORARY_ROOT(type, variable,value)
```

creates a local variable of the specified type with the specified initial value. The value must either be a pointer to an object in the heap, or it must be a value that is clearly not in the heap (such as `NULL`, a pointer to permanent space, or the like).[1] The value `&variable` is registered with the garbage collector as a temporary root.

You are allowed to change the value of `variable`, provided that any new value is always either a pointer to an object in the heap, or a value that is clearly not in the heap.

The garbage collector ensures whenever a garbage collection occurs, the value of the variable is updated if the value has moved. In addition, `&variable` is a handle, and can be passed as an argument to any function that expects a handle.

Your code must not `return`. The `END_TEMPORARY_ROOTS` contains cleanup code that must be executed.

CODE EXAMPLE 6 below shows some sample code for a native method that takes a `String` and two integers as arguments, and which must allocate a temporary buffer.

CODE EXAMPLE 6 Temporary roots

```
START_TEMPORARY_ROOTS
    int y = popStack();
    int x = popStack();
    DECLARE_TEMPORARY_ROOT(STRING_INSTANCE, string,
                           popStackAsType(STRING_INSTANCE));
    DECLARE_TEMPORARY_ROOT(char*, buffer, mallocBytes(100));
    /* code that might perform allocation */
END_TEMPORARY_ROOTS
```

If the code clearly cannot perform any allocation, then you could instead have written

```
char* buffer = mallocBytes(100);
```

Less commonly used is the macro

```
DECLARE_TEMPORARY_ROOT_FROM_BASE(type, var, value, base)
```

In this case *base* must be a pointer to an object in the heap, and *value* must be a pointer into the middle of the object. The variable `var` is assigned the value *value*. The garbage collector will treat *base* as a root. If *base* is moved by the garbage collector, the value of `var` will be adjusted appropriately.

1. The main purpose of this limitation is that the variable should not have a random integer as its value, and that the variable must be initialized.

11.2.8.5 Global roots

If your code initializes a C variable to point to an object in the Java heap, you can use the code shown in CODE EXAMPLE 7. There is currently no function for removing a variable from the set of global roots.

CODE EXAMPLE 7 Creating a global root

```
variable = <value>
makeGlobalRoot(&(cell **)variable);
```

This code ensures that the garbage collector knows that the specified variable contains a value that must be protected from garbage collection. If the garbage collector moves the object, the variable is updated to point to the new value.

11.2.8.6 Debugging your native code

A special garbage collector is provided to help you debug your native code and to ensure that it does not have any garbage collection problems. You access this garbage collector by replacing the file `collector.c` with `collectorDebug.c`. In addition, you should set the compiler flags `INCLUDEDEBUGCODE` and `EXCESSIVE_GARBAGE_COLLECTION` to 1.

This replaces the compact-in-place garbage collector with a 10-space Cheney style[1] garbage collection algorithm. A garbage-collection will occur on every allocation, and also on some operations that might have allocated but didn't. Every object moves on every garbage collection. In addition, this code makes use of memory-protection so that any attempts to read or write a bad pointer will generate a memory fault.

This code uses the following implementation-dependent functions:

```
void* allocateVirtualMemory_md(long size);
void  freeVirtualMemory_md(void *address, long size);
void  protectVirtualMemory_md(void *address, long size,
                              int protection);
```

Implementations of these three functions for Windows and for Unix are provided. You must implement these functions on your target platform.

11.2.8.7 Two-space Cheney garbage collector

The file `collectorDebug.c` (see §11.2.8.6) also includes an implementation of a two-space non-debugging Cheney garbage collector. You get this implementation by setting the compiler flag `CHENEY_TWO_SPACE` to a non-zero value.

1. C.J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677-8, November 1970.

The Cheney collector is smaller and faster than the standard garbage collector. However, it uses twice as much heap space. If your implementations has a lot of available memory, but needs a faster garbage collector, you might consider using this garbage collector.

This collector is not supported by Sun, and is provided as is.

11.2.9 Initialization and reinitialization of global variables

Generally, the C language guarantees that all global and static variables are initialized to `0` (zero).

The current implementation is designed to work within an embedded environment. For example, on the PalmOS, the user can start the virtual machine, exit a program, and then restart the virtual machine with a different set of arguments. There is no re-initialization of global or static variables between the two runs.

In general, your code cannot assume the initial value of any variable. You have several options for determining when it is necessary to perform one-time only initialization.

- You can use the function `InitializeNativeCode()` to either initialize your variables, or to set a flag indicating that initialization needs to be performed.
- If a private native method is called as part of static initialization of a class, the method's native implementation will be called the first time the class is used. The native implementation can perform any initialization necessary for the class.
- If a variable is part of the global root set (see `makeGlobalRoot()` above), its value is guaranteed to be 0 the next time that the virtual machine is run.

11.3 Native code lookup tables

Regardless of whether you use the KNI (Section 11.1 "Using the K Native Interface (KNI)") or the old-style native method implementation technology (Section 11.2 "Implementing old-style native methods"), as part of the build process you must create the lookup tables that map methods to the corresponding native implementation.

The JavaCodeCompact (JCC) generates these tables automatically. You should use this utility to generate the lookup tables whether or not you are using the other features of JavaCodeCompact.

JavaCodeCompact is more fully described in Chapter 14. The specific details for creating the file containing the lookup tables can be found in Section 14.5 "Executing JavaCodeCompact."

The name of the C function that implements a native method must be the same name that JNI[1] would assign to the native method.

11.4 Asynchronous native methods

Note – The KNI implementation does not allow asynchronous native methods to be written using the KNI API. In order to write asynchronous native methods for the KVM, you must use the old-style native function interface as described in this section.

From the operating system viewpoint, KVM is just one process (C program) with only one native thread of execution. The multithreading capabilities of KVM have been implemented entirely in software without utilizing the possible multitasking capabilities of the underlying operating system. This approach not only makes the virtual machine highly portable and independent of the operating system, but also greatly simplifies the virtual machine design and improves the readability of the codebase, as the virtual machine designer does not have to worry about mutual exclusion and other problems typically associated with multithreaded software.

However, an unfortunate side effect of the approach described above is that by default, all native methods in KVM are "blocking." This means that when a native function is called from the virtual machine, all the threads in the VM stop executing until the native method completes execution.

As a general guideline, all the native functions called from KVM should be written so that they complete their execution as soon as possible. However, in many environments this is not desirable or fully possible. For this reason, KVM includes an implementation of "asynchronous native methods" described below.

11.4.1 Design of asynchronous methods

The standard implementation of KVM runs as a single "task" from the operating system's point of view. If a native method performs an operation that can block, the entire KVM blocks.

1. See *The Java Native Interface: Programmer's Guide and Specification (Java Series)* by Sheng Liang (Addison Wesley, 1999), for complete information on the JNI naming scheme. This information is available online at `http://java.sun.com/docs/books/jni/index.html`

Asynchronous native methods are intended to solve this problem. When such a native method is called, the operation is performed “off-line” in an implementation-dependent manner. Other Java threads can continue running normally. When the native call finishes, the Java thread that originally called the native method continues.

To use asynchronous native methods, you must include

```
#define ASYNCHRONOUS_NATIVE_FUNCTIONS 1
```

in your machine-dependent include file.

Asynchronous native methods cannot be defined in the same file as normal native methods. In addition to their normal includes, they must also add the include file `async.h`.

Asynchronous methods should always have the following form:

```
ASYNC_FUNCTION_START(functionname)
    code
ASYNC_FUNCTION_END
```

Your code must never use `pushStack()`, `popStack()`, `topStack`, or any macro or function that references the stack pointer, the frame pointer, or the current thread. Instead, you must use the alternative macros shown in Table 14.

TABLE 14 Macros used in asynchronous methods

Native function macro	Asynchronous native function macro
`popStack`	`ASYNC_popStack`
`pushStack`	`ASYNC_pushStack`
`popLong`	`ASYNC_popLong`
`pushLong`	`ASYNC_pushLong`
`popStackAsType`	`ASYNC_popStackAsType`
`pushStackAsType`	`ASYNC_pushStackAsType`
`raiseException`	`ASYNC_raiseException`
`topStack`	do not use this macro

In addition, your code must not perform a “return.” It must complete through the end, since `ASYNC_FUNCTION_END` may generate some necessary cleanup code.

All the macros in Table 14 have been designed so that if the symbol `ASYNCHRONOUS_NATIVE_FUNCTIONS` is 0, the asynchronous method compiles into a normal native method.

It is also important to note that unlike regular native methods, asynchronous native methods *cannot allocate any memory from the Java heap*. Because of this limitation, extra caution is often necessary when writing asynchronous native methods, since many internal routines in KVM may indirectly allocate memory from the Java heap.

Note – IMPORTANT: We repeat that asynchronous native methods *must not* allocate memory from the Java heap. Make sure that you read the paragraph above.

If you use asynchronous native methods, you must define the following machine-specific functions.

- `void Yield_md()`

  Pause this operating system task momentarily and allow other tasks to run.

- `void CallAsyncNativeFunction_md(ASYNCIOCB *iocb, void(*afp)(ASYNCIOCB *))`

  Call an asynchronous native function. This function is called by the `ASYNC_FUNCTION_START` macro to start a new asynchronous function. The function takes as a parameter a data structure that is used by the garbage collector to keep up to date object pointers used by the native code, and a function to call. This function will typically start a new native thread and have that call the supplied function with the `ASYNCIOCB` as its parameter.

- `enterSystemCriticalSection()`
  `exitSystemCriticalSection()`

  Enter or exit a critical section. The operating system must guarantee that at most one operating system task is allowed to be inside the critical section at a time.

11.4.2 Implementation of asynchronous methods

We envision two possible implementations of asynchronous methods.

In the current reference implementation, the function `CallAsyncNativeFunction_md` spawns off a separate operating system task which performs the indicated function. For example, in a Posix implementation one could use `pthread_create`.

CODE EXAMPLE 8 below shows one possible implementation of a method `int readBytes(byte[] dst, int offset, int length)` using this style of asynchronous native methods. This particular example assumes that the garbage collector does not move objects.

CODE EXAMPLE 8 Asynchronous implementation of `ReadBytes`

```
ASYNC_FUNCTION_START(ReadBytes)
    long   length = ASYNC_popStack();
    long   offset = ASYNC_popStack();
    BYTEARRAY dst = ASYNC_popStackAsType(BYTEARRAY);
    INSTANCE instance = ASYNC_popStackAsType(INSTANCE);/* this*/
    long fd = getFD(instance);
    ASYNC_enableGarbageCollection();
    length = read(fd, dst->bdata + offset, length);
    ASYNC_disableGarbageCollection();
    ASYNC_pushStack((length == 0) ? -1 : length);
ASYNC_FUNCTION_END
```

In an alternative implementation (CODE EXAMPLE 9), `CallAsyncNativeFunction_md` simply calls the function `f` directly. It assumes that the function `f` starts an operation, but does not wait for its completion. The operating system is required to provide some sort of interrupt or callback to indicate when the operation is complete.

The second implementation is far more operating system-dependent. It might be impossible to write native methods that can work both synchronously and asynchronously, depending on the value of a flag.

CODE EXAMPLE 9 Alternative asynchronous implementation of `ReadBytes`

```
static void ReadBytes(THREAD thisThread)
{
    long   length = ASYNC_popStack();
    long   offset = ASYNC_popStack();
    BYTEARRAY dst = ASYNC_popStackAsType(BYTEARRAY);
    INSTANCE instance = ASYNC_popStackAsType(INSTANCE);
    long fd = getFD(instance);
    THREAD thisThread = CurrentThread;
    /* Call OS to perform I/O. Perform callback when done. */
    AsyncRead(fd, p + offset, length, ReadBytesDone,thisThread);
}

/* Callback function when I/O is finished */
static void ReadBytesDone(void *parm, int length)
{
    THREAD thisThread = (THREAD)parm;
    ASYNC_pushStack((length == 0) ? -1 : length);
    ASYNC_RESUME_THREAD();
}
```

Refer to Section 12.1.4 “Asynchronous notification,” for further information on writing asynchronous code.

CHAPTER 12

Event Handling

12.1 High-level description

The Java Virtual Machine Specification does not define how the virtual machine interacts with events that arrive from the host operating system or from the target device. The KVM implementation, however, provides a variety of mechanisms that were designed to facilitate the integration of the KVM with the event system mechanisms of the host operating system or device.

There are four ways in which notification and handling of events can be accomplished in KVM:

1. Synchronous notification (blocking).
2. Polling in Java code.
3. Polling in the bytecode interpreter.
4. Asynchronous notification.

Different solutions may be appropriate for different ports of the KVM, depending on which user interface libraries are supported, what kinds of networking libraries are used, and so forth.

12.1.1 Synchronous notification (blocking)

By synchronous notification we refer to a situation in which the KVM performs event handling by calling a native I/O or event system function directly from the virtual machine. Since the KVM has only one physical thread of control inside the virtual machine, no other Java threads can be processed while the native function is being executed, and no VM system functions such as garbage collection can occur either. This is the simplest form of event notification, but there are many situations

in which this solution is quite acceptable, provided that the person designing the native functions is careful enough to keep the native functions as short and efficient as possible.

For instance, writing a datagram into the network can typically be performed efficiently using this approach, since typically the datagram is sent to a network stack that contains a buffer and the time spent waiting for the event to complete is very small. In contrast, reading a datagram is often a very different story, and is often handled better using the other solutions described below. Using a native function to wait until a whole datagram is received would block the whole KVM while the read operation is in progress.

12.1.2 Polling in Java code

Often event handling can be implemented efficiently using a combination of native and Java code. This is a simple way to allow other Java threads to execute while waiting for an event to complete. When using this approach, a polling Java loop is normally put somewhere in the Java runtime libraries so that the loop is hidden from applications. The normal procedure is for the runtime library to initiate a short native I/O operation and then repeatedly query the status of the I/O operation until it is finished. The polling Java code loop should always contain a call to `Thread.yield` so that other Java threads can be allowed to run efficiently.

This method of waiting for event notification is very easy to implement and is free of any complexities typically associated with genuinely asynchronous threads (such as requiring critical sections, semaphores or monitors.) There are two disadvantages with this design. First, CPU cycles are needed to perform the Java-level polling that could otherwise be used to run application code (although the overhead is usually very small.) Second, due to the interpretation overhead, there may be some extra latency associated with event notification (especially if you forget to call `Thread.yield` in the polling Java code loop.) Again, this overhead is usually negligible in all but most time-critical applications.

12.1.3 Polling in the bytecode interpreter

The third approach to implement event handling is to use the bytecode interpreter periodically make calls to the native event handling operations. This approach is a variation of the synchronous notification approach described above. This approach was originally used extensively in the KVM, for example, to implement GUI event handling for the Palm platform.

In this approach, a native event handling function is called periodically from the interpreter loop. For performance reasons this is not normally done before every bytecode, but every few hundred bytecodes or so. This way the cost of performing event handling is well amortized. By changing the number of bytecodes executed before calling the event handling code, the virtual machine designer can control the

latency of event delivery versus the CPU time spent looking for a new event. The smaller the number, the smaller latency and the larger CPU overhead. A large number reduces CPU overhead but increases the latency in event handling.

The advantage of this approach is that the cost in performance is less than polling in Java, and the event notification latency is more predictable and controllable. The way this approach works is closely related to asynchronous notification described in the next subsection.

12.1.4 Asynchronous notification

The original KVM implementation supported only the three event handling implementations discussed above. However, in order to support truly asynchronous event handling, some new mechanisms have been introduced.

By asynchronous notification we refer to a situation in which event handling can occur in parallel while the virtual machine continues its execution. This is generally the most efficient event handling approach and will typically result in a very low notification latency. However, this approach generally requires that the underlying operating system provides the appropriate facilities for implementing asynchronous event handling. Such facilities may not be available in all operating systems. Also, this approach is quite a bit more complex to implement, as the virtual machine designer must be aware of possible locking and mutual exclusion issues. The reference implementation provides some examples that can be used as a starting point when implementing more device-specific event handling operations.

The general procedure in asynchronous notification is as follows. A thread calls a native function to start an I/O operation. The native code then suspends the thread's execution and immediately exits back to the interpreter loop, letting other threads continue execution. The interpreter then selects a new thread to run. Some time later an asynchronous event occurs and as a result some native code is executed which resumes the suspended thread. The interpreter then restarts the execution of the thread that had been waiting for an event to occur.

At the implementation level, there are two ways to implement such asynchronous notification. One is to use native (operating system) threads, and the other is to use some kind of software interrupt, callback routine or a polling routine.

In the first case, before the native function is called and the Java thread is suspended, a new operating system thread is created (or reawakened) and it is this thread which enters the native function. There is now an additional native thread of control running inside the virtual machine. After the native I/O thread is started, the order of execution inside the virtual machine is no longer fully deterministic, but depends on the occurrence of external events. Typically, the original thread starts executing another Java thread in the interpreter loop, and the new thread starts the I/O operation with what is almost always a blocking I/O operation to the operating system.

It is important to note that the native I/O function will execute out of context meaning that the context of the virtual machine will be a different thread. A special set of C macros were written that will hide this fact for the most part, but special care should be taken to be sure that no contextual pointers are used in this routine. When the blocking call is finished the native I/O thread resumes execution and unblocks the Java thread it was representing. The Java thread is then rescheduled, and the native I/O thread is either destroyed, or placed in a dormant state until it needs to be used again. The Win32 port of the KVM reference implementation does this by creating a pool of I/O threads that are reused when I/O is to be performed.

The second implementation of asynchronous event handling can be done by utilizing callback functions associated with I/O requests. Here the native code is entered using the normal interpreter thread, I/O is started and then when the I/O operation is completed a callback routine is called by the operating system and the Java thread is unsuspended. In this scenario the native code is split into two routines, the first being a routine that starts the I/O operation and the second invoked when I/O is completed. In this case the first routine runs in the context of the calling Java thread, and the second one does not.

The final, less efficient variation of asynchronous event handling is where the I/O routine is polled for completion by the interpreter loop. This is very similar to the callback approach except that the second routine is called repeatedly by the interpreter to check if the I/O has finished. Eventually when the I/O operation has completed the routine unblocks the waiting Java thread. This calling of the native code by the interpreter is always done even when there are no pending events, and the native code must determine what Java threads should be restarted.

Synchronization issues. It is very important to remember that in the cases where a separate native event handling thread or callback routine is used, the code for event handling may interrupt the virtual machine at any point. Therefore, the person porting the virtual machine must remember to add critical sections, monitors or semaphores to all locations where the program may be manipulating common data structures and a possible mutual exclusion problem might occur. The most obvious shared data structures are the queues of suspended and active Java threads. These are always manipulated using special routine in the virtual machine that is already properly synchronized. If there are any other shared data structures they must be synchronized in the native code. Failure to do this correctly will produce spurious bugs that are very hard to debug.

12.2 Parameter passing and garbage collection issues

When native event handling code is called, its parameters will be on the stack for the calling Java thread. These are popped off the stack by the native code, and the if there is a result value to be returned this is pushed onto the Java stack just prior to resuming the execution of the thread. Native parameter passing issues are discussed in Chapter 11.

Because native event handling code can access object memory, there are possible garbage collection issues especially when running long, asynchronous I/O operations. In general, the garbage collector is prevented from running when there is any native code is running. This is a problem when certain long I/O operations are performed. The most obvious case is waiting for a incoming network request. To solve this problem two functions called `decrementAsyncCount` and `incrementAsyncCount` are provided. The first allows the garbage collector to start, and the second prevents the collector from starting, and waits for it to stop if it was running.

It should be noted that if an object reference is passed to a native method, but no other reference to it exists in Java code after the call to `incrementAsyncCount`, the object could be reclaimed accidentally by the garbage collector. It is hard to think of a realistic scenario where this could occur, but the possibility should be kept in mind. A possible example of such code is the following:

```
native read(byte[]);
void skipBytes(int n) {
    read(new byte[n]);
}
```

Here the only reference to the byte array object exists on the parameter stack to the native function. If the native code calls `incrementAsyncCount` after popping the parameter from the stack the array could be garbage collected.

12.3 Implementation in KVM

The event handling implementation in KVM is composed of two main layers that both need to be taken into account when porting the KVM onto new hardware platforms.

At the top of the interpreter loop is the following code (starting from KVM 1.0.2, this code is actually located in macros):

```
if (isTimeToReschedule())
        reschedule();
```

The standard rescheduling code performs the following operations.

1. Checks to see if there are any active Java threads and stops the VM if there are none.
2. Checks to see if enough time has passed to allow a thread that was waiting for a specific time to be restarted. If there is such a thread, it is automatically restarted.
3. Checks to see if any I/O events have occurred and where appropriate it allows the relevant threads to contend for CPU time
4. Attempts to switch to another thread.

For performance reasons, the operations above are implemented as macros that are, by default, defined in `VmCommon/h/events.h`. It is here that device-specific event handling code can be placed. By default, the `isTimeToReschedule` macro decrements a global counter and tests for it being zero. When it is zero the second macro is executed. The idea here is for the `reschedule` to be executed only once for a fairly large number of bytecode executions. As the name implies, `reschedule` is where the thread context switching is done, if necessary.

The second layer in event handling implementation is the function

```
GetAndStoreNextKVMEvent(bool_t forever, ulong64 waitUntil)
```

If a new event is available from the host operating system, this function must call a special function called `StoreKVMEvent` to make the details of the event available to the KVM. If no new events are available from the host operating system, then the function can simply return.

The arguments to the `GetAndStoreNextKVMEvent` function are as follows:

- If the forever argument is `TRUE`, this function should wait for as long as necessary for an event to occur (used for battery conservation as described below.)
- If the forever argument is `FALSE`, this function should wait until at most `waitUntil` for an event to occur.

Some battery conservation features were included in the reference implementation of these functions. This is to pass to the event checking function the “forever” flag or the maximum wait time. If there are no pending events, the native implementation of the `GetAndStoreNextKVMEvent` function can then put the device “to sleep” until the next event occurs. Battery conservation issues are discussed in more detail in the next subsection.

12.4 Battery power conservation

Most KVM target devices are battery-operated, and the manufacturers of these devices are typically extremely concerned of excessive battery power consumption. To minimize battery usage, KVM is designed to stop the KVM interpreter loop from running whenever there are no active Java threads in the virtual machine and when the virtual machine is waiting for external events to occur. This requires support from the underlying operating system, however.

In order to take advantage of the power conservation features, you must port the following low-level event reading function

```
GetAndStoreNextKVMEvent(bool_t forever, ulong64 waitUntil)
```

so that it calls the host system specific sleep / hibernation features when the virtual machine calls this function with the `forever` argument set `TRUE`. The KVM has been designed to automatically call this function with the `forever` argument set `TRUE` if the virtual machine has nothing else to do at the time.

This allows the native implementation of the event reading function to call the appropriate device-specific sleep / hibernation features until the next native event occurs.

Additionally, the macro `SLEEP_UNTIL(wakeupTime)` should be defined in such a fashion that the target device goes to sleep until `wakeupTime` milliseconds has passed.

CHAPTER 13

Class File Verification

13.1 Overview

The class file verifier supported by Java 2 Standard Edition (J2SE) is not suitable for small, resource-constrained devices. The J2SE verifier requires a minimum of 50 kB binary code space, and at least 30-100 kB of dynamic RAM at run time. In addition, the CPU power needed to perform the iterative dataflow algorithm in the standard JDK verifier can be substantial.

We have designed and implemented a new, two-phase class file verifier that is significantly smaller than the existing J2SE verifier. The runtime part of the new verifier requires about 15 kB of Intel x86 binary code and only a few hundred bytes of dynamic RAM at run time for typical class files. The runtime verifier performs a linear scan of the byte code, without the need of a costly iterative dataflow algorithm. The new verifier is especially suitable for KVM, a small-footprint Java virtual machine for resource-constrained devices.

The new class file verifier operates in two phases, as illustrated in Figure 1:

- First, Java class files have to be run through a special *preverifier* tool in order to augment the class files with additional attributes to speed up runtime verification. The preverification phase is typically performed on a development workstation, where the application developer writes and compiles the applications.
- At runtime, the runtime verifier component in the KVM utilizes the additional attributes generated by the preverifier to perform the actual class file verification efficiently.

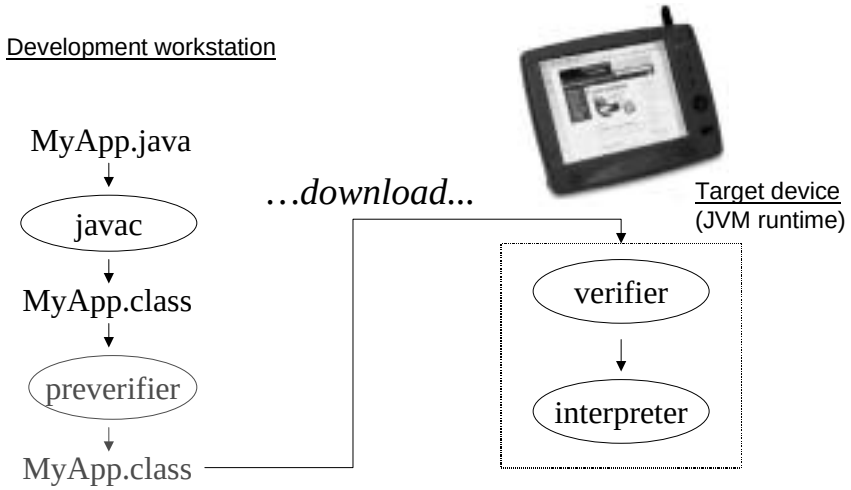

FIGURE 1 Two-phase verification

The runtime class file verifier requires all the subroutines to be inlined, so that class files contain no `jsr`, `jsr_w`, `ret`, or `wide ret` instructions. Additionally, the runtime verifier requires the methods in class files to contain special `StackMap` attributes. The preverifier tool performs these modifications to normal class files generated by a Java compiler such as `javac`. A transformed class file is still a valid J2SE class file, but with additional attributes that allow verification to be carried out efficiently at run time.

Note – In the future, `javac` (the Java compiler) may be modified to perform these changes automatically. In that case, the preverifier tool will no longer be necessary.

The preverifier tool shipped with the KVM release is a C program that contains code extracted from the JDK 1.1.8 virtual machine implementation as well as code specifically written for inlining subroutines and inserting the `StackMap` attributes. The program compiles and runs on Windows and Solaris, and can be ported to other development platforms relatively easily.

13.2 Using the preverifier

The preverification phase is usually performed at application development time on a development workstation. For example, if you weren't using a preverifier, you would typically compile `Foo.java` using `javac` like this:

```
javac -classpath kvm/classes Foo.java
```

However, when using the preverifier, you would place the output of `javac` in a separate directory and then transform the resulting class files using the preverifier. For example:

```
javac -classpath kvm/classes -d mydir Foo.java
preverify -classpath kvm/classes -d . mydir
```

The above preverifier command transforms all class files under `mydir/` and places the transformed class files in the current directory (as specified by the `-d` option).

Makefiles in the KVM distribution invoke the preverifier automatically.

13.2.1 General form

More generally, the preverifier is invoked as follows:

```
preverify <options> <input files>
```

Preverifier options and accepted input file formats are explained in more detail below.

13.2.2 Preverifier options

The preverifier accepts a number of arguments and options.

`-classpath <directories> | <JAR files>`

- Directories or JAR file(s) in which the KVM/CLDC Java library classes are located. The directory separator is platform-specific. On Solaris a colon is used. On Win32 a semicolon is used. The JAR file specified must be in a valid Java Archive format and must end with either `.jar`, `.JAR`, `.zip` or `.ZIP` suffix.

`-d <directory>`

- The directory in which output classes will be written. The default output directory is `./output`.

`-cldc1.0`

- This option checks for the existence of language features prohibited by CLDC 1.0 (native methods, floating point, and finalizers).

`-nofinalize`

- This option checks for the use of finalizers in application classes. When this option is specified, an error is reported if finalizers are detected in any of the input files.

`-nonative`

- This option checks for the use of native methods in application classes. When this option is specified, an error is reported if native methods are detected in any of the input files.

`-nofp`

- This option checks for the use of floating point operations in application classes. When this option is specified, an error is reported if floating point operations are detected in any of the input files.

`@<filename>`

- The name of a text file from which command line arguments will be read.

Note – When the command line arguments are read from a file, parameters must all be specified on a single line and the parameters to the `-classpath` and `-d` options must be enclosed within double quotes. When the corresponding options are used from the command line, quotes are not required (unless the directory / file name parameter contains spaces.)

For example, the contents of `<filename>` under Win32 may appear as follows:

```
-classpath “api/classes; aaa bbb ccc/samples/classes” -d “output”
    -verbose HelloWorld1 HelloWorld2 HelloWorld3
```

13.2.3 Supported input file formats

The preverifier can accept input files in three different formats:

- individual Java class files
- directories containing Java class files
- JAR files containing Java class files.

To preverify a single class file or a number of class files, simply include the class file(s) after the command line options:

```
preverify -classpath kvm/classes File1 File2 ...
```

To preverify all the Java class files contained in a directory or set of directories, invoke the preverifier tool as follows:

```
preverify -classpath kvm/classes dir1 dir2 ...
```

To preverify all the Java class files contained in one or more JAR files, invoke the preverifier tool as follows:

```
preverify -classpath kvm/classes Jar1.jar Jar2.jar ...
```

Any combination of individual class files, directories or JAR files should be possible.

Obviously, the library classes can also be contained in a JAR / ZIP file, as illustrated by the line below:

```
preverify -classpath classes.zip File1 File2 ...
```

Output is generated differently depending on input parameters. If individual files are specified, the preverifier tool performs preverification separately for each input file. For each directory name, the preverifier recursively transforms every class file under that directory. The JAR file handling is discussed in the next section.

Note – A non-zero error status is returned if preverification fails for any reason.

13.2.4 JAR support in preverifier (since KVM 1.0.2)

Since KVM 1.0.2, the preverifier tool provided with the KVM allows input files to be provided as a JAR file. Given a JAR file that contains un-preverified Java class files, the preverifier tool will automatically generate an identical JAR file containing preverified class files.

This is performed as follows: First, the preverifier will check the file extension (`.jar`, `.JAR`, `.zip` or `.ZIP` file suffixes are acceptable) and validate that the file is in valid Java Archive format. Then, the class files will be extracted from the JAR file. For each class name extracted from the JAR file, the preverifier tool will perform the necessary transformations, and will then store the output file into a temporary directory `tmpdir`. After all the class files have been transformed successfully, a new JAR file with the same name will be created under `<output>` directory containing all the verified classes previously stored in `tmpdir`.

If the preverifier is run in non-verbose mode, any errors that may have occurred during the JAR creation will be logged in the `<output>/jarlog.txt` file, where `<output>` refers to the directory in which output classes will be written. If no errors occur during JAR creation, the `<output>/jarlog.txt` file will be removed. Directory `tmpdir` is also removed after the JAR file creation.

Note – When preverifying class files contained in JAR files, the preverifier tool will internally call the standard JAR tool to repackage the output files into a new JAR file. To accomplish this, the standard JAR tool must be accessible on your file path.

13.3 Porting the verifier

Runtime part. The runtime part of the verifier does not generally require any porting efforts, as it is closely integrated with the rest of the KVM, and is implemented in portable C code.

The process of porting the runtime verifier from KVM to another virtual machine is beyond the scope of this document.

Preverifier part. The preverifier is also written in C. By default, the preverifier is available for Windows and Solaris, but it should be relatively easy to compile it to run on other operating systems as well. Note that the preverifier codebase is derived from the “Classic” Java virtual machine, so the preverifier implementation looks quite different from the rest of the KVM codebase.

13.3.1 Compiling the preverifier

The sources for the preverifier are in the directory `tools/preverifier/src`.

On Solaris, you can build the preverifier by typing the “gnumake” command in the `tools/preverifier/build/solaris` directory. This compiles and links all `.c` files in the `tools/preverifier/src` directory, and places the resulting executable file in the `tools/preverifier/build/solaris` directory.

On Win32, you can build the preverifier by typing the “gnumake” command in the the `tools/preverifier/build/win32` directory. This compiles and links all `.c` files in the `tools/preverifier/src` subdirectory, and places the resulting executable file in the `tools/preverifier/build/win32` directory.

CHAPTER 14

JavaCodeCompact (JCC)

KVM supports the *JavaCodeCompact* (JCC) utility (also known as the class *prelinker, preloader* or *ROMizer*). This utility allows Java classes to be linked directly in the virtual machine, reducing VM startup time considerably.

At the implementation level, the JavaCodeCompact utility combines Java class files and produces a C file that can be compiled and linked with the Java virtual machine.

In conventional class loading, you use javac to compile Java source files into Java class files. These class files are loaded into a Java system, either individually, or as part of a jar archive file. Upon demand, the class loading mechanism resolves references to other class definitions.

JavaCodeCompact provides an alternative means of program linking and symbol resolution, one that provides a less-flexible model of program building, but which helps reduce the VM's bandwidth and memory requirements.

JavaCodeCompact can:

- combine multiple input files
- determine an object instance's layout and size
- load only designated class members, discarding others.

14.1 JavaCodeCompact options

JavaCodeCompact accepts a large number of arguments and options. Only the options currently supported by KVM are given below.

- `filename`

  Designates the name of a file to be used as input, the contents of which should be included in the output. File names with a `.class` suffix are read as single-class files.

  File names with `.jar` or `.zip` suffixes are read as Zip files. Class files contained as elements of these files are read. Other elements are silently ignored.

- `-o output filename`

  Designates the name of the output file to be produced. In the absence of this option, a file is produced with the name `ROMjava.c`.

- `-nq`

  Prevents JavaCodeCompact from converting the byte codes into their "quickened" form. This option is currently required by KVM.

- `-classpath path`

  Specifies the path JavaCodeCompact uses to look up classes. Directories and zip files are separated by the delimiting character defined by the Java constant `java.io.File.pathSeparatorChar`. This character is usually a colon on the Unix platform, and a semicolon on the Windows platform.

  Multiple classpath options are cumulative, and are searched left-to-right. This option is used in conjunction with the `-c` cumulative-linking option, and with the `-memberlist` selective-linking option.

- `-memberlist filename`

  Performs selective loading as directed by the indicated file. This file is an ASCII file, as produced by JavaFilter, containing the names of classes and class members.

- -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 only of interest as a debugging aid.

- `-arch` *Architecture*

  Specify the architecture for which you are generating a romized image. At this time, you must specify `KVM` as the architecture.

14.2 Porting JavaCodeCompact

With one exception, JavaCodeCompact outputs C code that is completely platform-independent.

To initialize a variable that is `final static long` or `final static double`, JavaCodeCompact performs the appropriate initialization using the two macros:

```
ROM_STATIC_LONG(high-32-bits, low-32-bits)
ROM_STATIC_DOUBLE(high-32-bits, low-32-bits)
```

If you have initialized either the compiler `BIG_ENDIAN` or `LITTLE_ENDIAN` to a non-zero value, the file `src/VmCommon/h/rom.h` generates default values for these macros.

If you have not defined BIG_ENDIAN or LITTLE_ENDIAN, or if for some reason the macros defined in rom.h are inappropriate for your platform, you should create appropriate definitions for ROM_STATIC_LONG and / or ROM_STATIC_DOUBLE in a platform-dependent location.

There are no other known platform or port dependencies.

14.3 Compiling JavaCodeCompact

The sources for JavaCodeCompact are in the directory tools/jcc/src.

On Unix and Windows machines, you compile JavaCodeCompact by typing the command "gnumake" in the tools/jcc/ directory. This compiles all .java files in the tools/jcc/src subdirectory, and places the resulting compiled file in the tools/jcc/classes directory.

You may need to make modifications to the Makefile in the tools/jcc/ directory to indicate the location of your javac compiler.

14.4 JavaCodeCompact files

The directory tools/jcc contains a Makefile that shows all the steps necessary to execute JavaCodeCompact. This Makefile currently has two targets:

```
unix
windows
```

each of which can be used to create all the files necessary for that platform.

On the unix and windows platforms, two files are created:

```
ROMjavaPlatform.c
nativeFunctionTablePlatform.c
```

The first file contains the C data structures that correspond to the classes in the zip file. The second file contains tables necessary for using native functions (see §11.3). This second file should be compiled and linked into KVM whether or not you are planning to use the other features of the JavaCodeCompact utility.

14.5 Executing JavaCodeCompact

The JavaCodeCompact utility is used to built the platform-specific file `nativeFunctionTablePlatform.c`, which contains tables necessary for calling native methods.

This file must be built even if you are not using the ability of JavaCodeCompact to pre-load classes for you. If you are not ROMizing your system classes (in other words, you are loading all system classes dynamically), you may skip Step 4 below.

The simplest method for using the JavaCodeCompact utility is to either use the Makefile provided or to modify it for your platform. The following lists the steps that the Makefile performs:

1. Compile all the .java files in the `api/src` directory. The resulting class files are verified and merged into a single zip file `classes.zip`. This zip file is copied to the `tools/jcc` directory.

2. Compile the sources for JCC as described in §14.3 above.

3. Copy `classes.zip` to `classesPlatform.zip`. Remove from this platform-dependent zip file any classes or packages that should not be used on your platform.

4. Execute your system's equivalent of the following command in the `jcc` directory:

```
env CLASSPATH=classes \
     JavaCodeCompact -nq -arch KVM \
     -o ROMjavaPlatform.c classesPlatform.zip
```

   The "`env CLASSPATH-classes`" sets an environment variable indicating that the code for executing JavaCodeCompact can be found in the subdirectory called `classes`. Next on the command line is the name of the class whose main method is to be executed (`JavaCodeCompact`), and the arguments to that method.

5. Execute your system's equivalent of the following command in the `jcc` directory:

```
env CLASSPATH=classes \
     JavaCodeCompact -nq -arch KVM_Native
     -o nativeFunctionTablePlatform.c classesPlatform.zip
```

   This command creates the file containing the native function tables necessary to link native methods to the corresponding C code.

6. Recompile all the sources for KVM. You must ensure that the preprocessor macro `ROMIZING` is set to a non-zero integer value. You must also ensure that the file `ROMjavaPlatform.c` is included as one of your source files.

The resulting kvm image will include, pre-loaded, all of the class files that were in the original `classesPlatform.zip` file.

14.6 Limitations

The current implementation of JavaCodeCompact requires that the class files that you compact constitute a "transitive closure." If class `A` is compacted, and class `A`'s constant pool references class B, then class `B` must also be included as part of the compaction.

Class A includes Class B in its constant pool if any of the following conditions are true:

- Class A is a direct subclass of class B, or class A directly implements class B.
- Class A creates an instance of class B, or an array of class B.
- Class A calls a method that is defined in class B.
- Class A checks to see if an object is an instance of type B, or casts an object to type B.

Note that the following do not cause class B to be included in class A's constant pool. Under certain circumstances, it may be possible to compact A without also compacting B.

- Class A has an instance variable of type B
- Class A has a method whose argument or return type includes type B in its signature.
- Class A creates an instance of class B using the `Class.forName()` method.

JavaCodeCompact will fail and give you an error message if you fail to include a class file that it requires.

CHAPTER 15

Java Application Manager (JAM)

A central requirement for KVM in most target devices is to be able to execute applications that have been downloaded dynamically from the network. Once downloaded, the user commonly wants to use the applications several times before deleting them. The process of downloading, installing, inspecting, launching and uninstalling of Java applications is referred to generally as *application management*. In typical desktop computing environments, these tasks can be performed by utilizing the facilities of the host operating system. However, the situation is very different in many small, resource-constrained devices which often lack even basic facilities such as a built-in file system.

To facilitate the porting of KVM to small, resource-constrained platforms, KVM implementation contains an optional component called *Java Application Manager* (JAM) that can be used as a starting point for machine-specific implementations.

Note – The JAM that is provided as part of the CLDC Reference Implementation is used primarily for compatibility testing purposes. This JAM implementation is not compatible with the requirements of J2ME profiles such as MIDP. To implement a MIDP-compliant Java Application Manager, refer to the MIDP Reference Implementation.

At the compilation level, JAM can be turned on or off by using the flag

```
#define USE_JAM 1
```

When building the KVM using `gnumake`, the following command automatically builds the system with the JAM enabled:

```
gnumake USE_JAM=true
```

This section provides a brief overview of the JAM reference implementation provided with KVM. The description below assumes that the target device has some kind of a “microbrowser” that can be used for initiating the downloading of applications. This microbrowser is commonly provided as part of the native computing environment, but it can also be part of the JAM in some implementations.

15.1 Using the JAM to install applications

Java Application Manager is a native C application that is responsible for downloading, installing, inspecting, launching, and uninstalling Java applications.

From the user's viewpoint, the JAM is typically used as follows:

1. The user sees an application advertised on a content provider's web page.
2. The user selects the tag to install it.
3. The Java application is downloaded and installed.
4. The user runs it.

Here's a more detailed description:

1. While browsing a content provider web page using a native microbrowser, the user sees a description of the Java application in the text of the page, and a highlighted tag (or button) that asks them if they want to install the application. The tag contains a reference to an application *Descriptor File*. The Descriptor File, typically with a `.jam` file extension, is a text file consisting of name/value pairs. The purpose of this file is to allow the JAM to decide, before it tries to download it, whether the Java application the user selected can be installed successfully on the device. This saves the user the cost of moving the Java application to the device if it cannot be installed. The Descriptor File is small (several hundred bytes), while a typical Java application is from 10 to 20 kilobytes, so it is much cheaper to download the Descriptor File rather than the entire Java application.
2. The user selects the tag to start the installation process. The browser retrieves the Descriptor File from the web site.
3. The browser transfers program control to the JAM, passing it the content of the Descriptor File and the URL for the page it was browsing.
4. The JAM checks to see if the application is already installed on the device, and checks its version number (see later discussion on the details of application updating.) It then reads the `JAR-File-Size` tag of the Java application to ensure that there is sufficient space on the device to save it.
5. If there is sufficient space to install the application, the JAM uses the `JAR-File-URL` tag in the descriptor file to get the URL of the JAR file (it may use the base URL to the Descriptor File, if the `JAR-File-URL` tag is a relative URL) and start the download process using HTTP. The JAM then stores the JAR file on the device.

   If the download process is interrupted, the JAM discards the partially downloaded application, as if the application was never downloaded before.

6. The JAM adds the application to the list of installed Java applications, and registers it with any other native tools as required. The JAM saves the following information along with the JAR file:

   - name of JAR file,
   - absolute URL from which the JAR file was downloaded,
   - main class of the java application,
   - name of the application,
   - version number of the application.

   The absolute URL and the version number are used to uniquely identify an application during application update (see next subsection.)

   In the reference implementation of the JAM, the user is shown the list of installed Java applications on the device, with the recently installed application selected for execution.

   However, if the `Use-Once` tag is set to yes, JAM does not add the application to the list, and it launches the application immediately.

7. Any errors encountered during the process must be handled by the JAM. A help page URL for the content provider is included in the Descriptor File. The JAM can then direct the user to this URL using the native browser.

15.1.1 Application launching

Here's a typical use case for launching a Java application:

1. The user is shown a list of Java applications (the user interface design is left up to the manufacturer.)
2. The user selects the Java application that is to be launched (the user interface design and selection mechanism is left up to the manufacturer).
3. The JAM launches the KVM with a parameter containing the `main` class of the application. The KVM initializes the `main` class and starts executing it. As additional classes are required for the execution of the application, the KVM uses a manufacturer-defined API to unpack and load the class files from the stored JAR file.
4. The Java application is displayed on the screen to the user.
5. When the application exits, and if the `Use-Once` tag in the Descriptor File is set to `YES`, the JAM removes the JAR file.

15.1.2 Application updating

When the content provider updates an application (for example, to fix bugs or add new features), the content provider should do the following:

1. Assign a new version number to the application.
2. Change the Descriptor File of the application to use the new version number.
3. Post the updated JAR file on the content provider's web site, using the same `JAR-File-URL` tag as the previous version of the application.

When the user requests the installation of an application, the JAM checks if the application's `JAR-File-URL` is the same as one of the installed applications. If so, and the Application-Version of the requested version is newer than the installed version, the JAM prompts for user approval before downloading and installing the newer version of the application.

The reference implementation uses a string to specify the version number in the following format:

`Major.Minor[.Micro] (X.X[.X])`, where the `.Micro` portion is optional (it defaults to “0”). In addition, each portion of the version number is allowed to a maximum of 2 decimal digits (that is, the range is from 0 to 99.)

For example, “1.0.0” can be used to specify the first version of an application. For each portion of the version number, leading zeros are not significant. For example, “08” is equivalent to “8”. Also, “1.0” is equivalent to “1.0.0”. However, “1.1” is equivalent to “1.1.0”, and not “1.0.1”.

In the reference implementation, missing Application-Version tag is assumed to be “0.0.0”, which means that any non-zero version number is considered as a newer version of the application.

The JAM must ensure that if the application update fails for any reason, the older version is left intact on the device. When the update is successful, the older version of the application is removed.

15.2 JAM components

15.2.1 Security requirements

The JAM, its data, and associated libraries, should be stored securely on the device. The device manufacturer must ensure that these components cannot be modified by Java applications or other downloadable content.

15.2.2 JAR file

JAR files are a standard feature of Java technology designed to hold class files and application resource data in a compressed format. JAM-compliant JAR files hold exactly one Java application and its associated resources. Compressed JAR files reduce the size of the application by approximately 40% to 50%. This both reduces the storage requirements on the device and reduces the download time for the application. Items in the JAR file are unpacked as required by the JAM.

15.2.3 Application Descriptor File

The Application Descriptor File is a readable text file. It consists of name-value pairs that describe the important aspects of its associated Java application. It is referenced from a tag on a content provider's web page. It is created and maintained by the Java application developer and stored along with its application JAR file on the same web site. Developers may create this file with any text editor.

The Descriptor File has the following entries (tag names are case sensitive):

`Application-Name`

Displayable text, limited to width of screen on the device

`Application-Version`

`Major.Minor[.Micro]` (`X.X[.X]`, where `X` is a 1 or 2 digit decimal number, and the `.Micro` part is optional)

`KVM-Version`

Comma separated list of KVM version strings as defined in the CLDC `microedition.configuration` system property (see CLDC Specification). “CLDC-1.0” is an example of the KVM version string. The items in the list are matched against the KVM version string on the device, and an exact match is required to execute this application. Any item matching the KVM version string on the device satisfies this condition. For example, “CLDC-1.0, CLDC-1.0.3” runs on either version of KVM on the device.

`Main-Class`

Text name of the application's Main class in standard Java format.

`JAR-File-Size`

Integer in bytes

`JAR-File-URL`

Standard URL text format to specify the source URL. If this is a relative URL, then the URL to the Descriptor File is the base URL.

`Use-Once`

yes/no

`Help-Page-URL`

Standard URL text format, used by the browser to access help pages

Additional requirements and restrictions:

- The MIME type for the Descriptor File is `application/x-jam` and the extension is `.jam`.
- All URLs must point to the same server from which the web page was loaded.
- The JAM must store the Descriptor File contents, in a manufacturer-specific format for possible later use.

The application developer may add any application specific name-value pairs to the Descriptor File. This allows the application to be configured at deployment by changing the values in the Descriptor File. So, different Descriptor Files could use the same application JAR file, with different application parameters.

The format of the tag is a string, but it is recommended that it follow a similar style as the tags defined in the above table. The format of the value is an application specific string.

A simple proposed API to retrieve the value via the JAM could be:

```
public String GetApplicationParameter(String name)
```

15.2.4 Network communication

Whenever a Java application tries to make an HTTP connection, the networking implementation should check with the JAM to find the name of the server where the application was downloaded. This ensures that the connection is made to the same server the application came from. A string comparison is made between the host name in both the URLs.

15.3 Application lifecycle management

The lifecycle of a Java application is defined to be the following:

- The KVM task is launched and instructed to execute the main class of the Java application (as described by the `Main-Class` entry of the Descriptor File.)
- The Java application executes inside the context of the KVM task and responds to user events.
- The KVM task exits, either voluntarily, or involuntarily, and terminates the Java application.

The term task is used loosely to describe the KVM as a logically distinct execution unit. In actual devices, the KVM task can be implemented as a task, a process or a thread of the underlying operating system.

The API functions for controlling the lifecycle of the KVM are not specified, as the mechanism is vastly different from platform to platform. Instead, it is required that all JAM implementations support the following features:

- The JAM implementation must be able to launch the KVM task and start executing the `main` class of the Java application.
- The JAM implementation must be able to forcibly terminate the KVM task, and optionally be able to suspend and resume the KVM task.
- The suspension, resumption, and termination of the KVM must be performed by the procedures described below.

15.3.1 Termination of the KVM Task

The KVM task can be terminated in two ways: voluntarily or involuntarily.

The application can voluntarily terminate itself by calling the Java method `System.exit`. Under certain conditions, the JAM may decide to force the KVM to terminate. The exact method of triggering forced termination is platform dependent. For example, the JAM may spawn a watchdog thread that wakes up after a certain period. If the watchdog thread detects that the KVM has not terminated voluntarily, it forces the KVM to terminate.

During forced termination, the JAM actively frees all resources allocated by the KVM and terminates the KVM task. The exact procedure is platform dependent. On some platforms, calling `exit` or `kill` may be enough. On other platforms, more elaborate clean-up may be required.

15.4 Error handling

The JAM is responsible for handling all errors encountered in installing and launching Java applications. The method of handling errors differs from implementation to implementation, but the JAM should be able to interact with the user to resolve the error if possible. To assist in this, the Descriptor File has a tag called `Help-Page-URL` that is set by the content provider. The JAM may decide

that under certain conditions, the browser should be invoked and the user sent to the help page. The help page could have information that would allow the user to contact the content provider for additional assistance.

15.4.1 Error conditions

The following are a set of possible error conditions and sample messages that can be displayed to describe the error to the user. Manufacturers should design the messages so that they are appropriate to their device user interface.

- The user tries to install an application whose size is larger than the total storage space available on the device:

  "NAMEOFAPP" is too large to run on this device and cannot be installed.

- The user tries to install an application, whose size is larger than the free storage space (but smaller than the total storage space) on the device:

  There is not enough room to install. Try removing an application and trying again.

- The user tries to install an application that is already installed on the device.

  "NAMEOFAPP" is already installed. (Soft buttons should be labeled OK and Launch. Launch would run the existing application on the device.)

- The user tries to install an application that is not designed for the particular device they own.

  "NAMEOFAPP" won't work on this device. Choose another application. (Soft button label = Back, Done.)

- The user tries to install an application and the tags describing the Java application have a syntax error or an invalid format that results in installation failure.

  The installation failed. Contact your ISP for help.

- The user tries to install an application, the URL to the application is incorrect or inaccessible, and the application cannot be installed.

  The URL for "NAMEOFAPP" is invalid. Contact your ISP for help.

- The user tries to install an application, the application is not the same size as described in the Descriptor File. The application should be discarded.

  "NAMEOFAPP" does not match its description and may be invalid. Contact your ISP for help.

- The user is installing an application. During application download, the connection drops, and the application is not loaded onto the device successfully.

  The connection dropped and the installation did not complete. Please try installing again. [Soft button label = Install, Back]

- The user is installing an application, and the URL specified matches exactly with the one located already on the device.

  The JAM should check the version # of both versions and present a decision to the user.

- The user tries to run an application and for some reason the application cannot launch (for example, the JAM failed to create a new OS task to run the KVM).

  Cannot launch “NAMEOFAPP”. Contact your ISP for help.

- The user has been running an application. The application tries to save to the scratchpad and fails.

  Cannot save data. Contact your ISP for help.

- The user is running an application and it crashes or hangs during execution. NOTE: This is a generic error.

  “NAMEOFAPP” has unexpectedly quit.

CHAPTER 16

Java-Level Debugging Support (KDWP)

KVM provides facilities for plugging the virtual machine into third-party Java development and debugging environments that are compliant with the *JPDA (Java Platform Debug Architecture)* specification supported by Java 2 Standard Edition. Further information on the JPDA architecture is available at `http://java.sun.com/products/jpda/`.

Due to strict memory constraints, KVM does not implement support for the JVMDI (Java Virtual Machine Debug Interface) and full JDWP (Java Debug Wire Protocol) specifications required by JPDA.

Instead, KVM implements a subset of the JDWP known as *KDWP (KVM Debug Wire Protocol)*. A specification of the KDWP protocol is available in a separate document listed in Section 1.2 “Related documentation.”

16.1 Overall architecture

The KDWP was designed to be a strict subset of the JDWP, primarily based on the resource constraints imposed on the KVM. In order to make KVM run with a JPDA-compatible debugger IDEs, a *debug agent (*debug proxy) program is interposed between the KVM and the JPDA-compatible debugger. The debug agent allows many of the memory-consuming components of a JPDA-compliant debugging environment to be located on the development workstation instead of the KVM, therefore reducing the memory overhead that the debugging interfaces have on the KVM and target devices. As obvious, the debugging interfaces can be turned off completely (at compile time) on those platforms and ports that do not need Java-level debugging support.

At the high level, the Java-level debugging support implementation consists of two parts:

- the actual code in the KVM to support a subset of the JDWP, and

- the debug agent that performs some of the debug commands on behalf of the KVM.

The overall architecture for the Java-level debugging interface is illustrated in Figure 2. In that figure, the topmost box represents the JPDA-compliant debugging environment (“JPDA Debugger”) running on a development workstation. The debugger is connected to the debug agent that talks to the KVM.

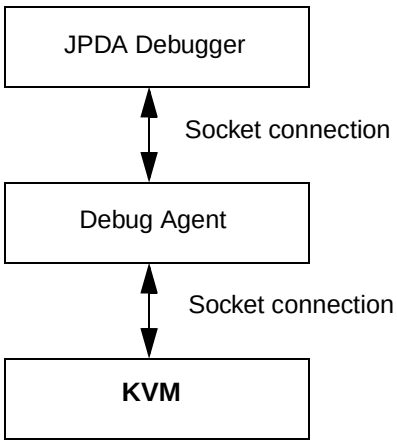

FIGURE 2 Java-level debugging interface architecture

The debug agent connects to the KVM via a socket connection. Similarly, the debugger connects to the debug agent over a socket. The debugger is unaware that it is connected to the debug agent. The debugger appears to be communicating directly with a JDWP-compliant Java Virtual Machine. In fact, the debug agent can be configured in *pass through mode* so that all packets are passed from input to output using the debug agent with a standard Java VM. In normal KVM debug mode, the debug agent examines packets from the debugger and determines which packets are to be handled by the KVM and which are to be handled within the debug agent.

The main processing done in the debug agent is the parsing of class files to extract debugging information. This includes line number and code offset information and variable information. The KDWP implementation within the KVM includes some *vendor specific commands* that the debug agent uses to communicate with the KVM.

16.2 Debug Agent

The debug agent *(*also known as *debug proxy)* is written in the Java programming language and the code is in the KVM source tree under the directory `tools/kdp/src/kdp`. There are two main portions of the code: the portion that handles connections to the debugger and to KVM, and the portion that handles the parsing of the class files. The latter code is located in subdirectory `classparser`.

16.2.1 Connections between a debugger and the KVM

The portion of the code that handles connections to the debugger and to KVM resides in file `KVMDebugProxy.java`. This code creates two objects: *`DebuggerListener`* and *`KVMListener`*. The `DebuggerListener` class handles the retrieval of packets from the debugger, and the `KVMListener` class handles the retrieval of packets from the KVM. `DebuggerListener` and `KVMListener` are both subclasses of class `Thread`. Therefore, when they are invoked they start a new thread of execution (on the development workstation.) Each object also gets passed a handle to the other object (that is, the `KVMListener` object gets passed a handle to the `DebuggerListener` object, and vice versa). This enables cross-communication of packets between the debugger and the KVM. The following diagram (Figure 3) may help to clarify this further:

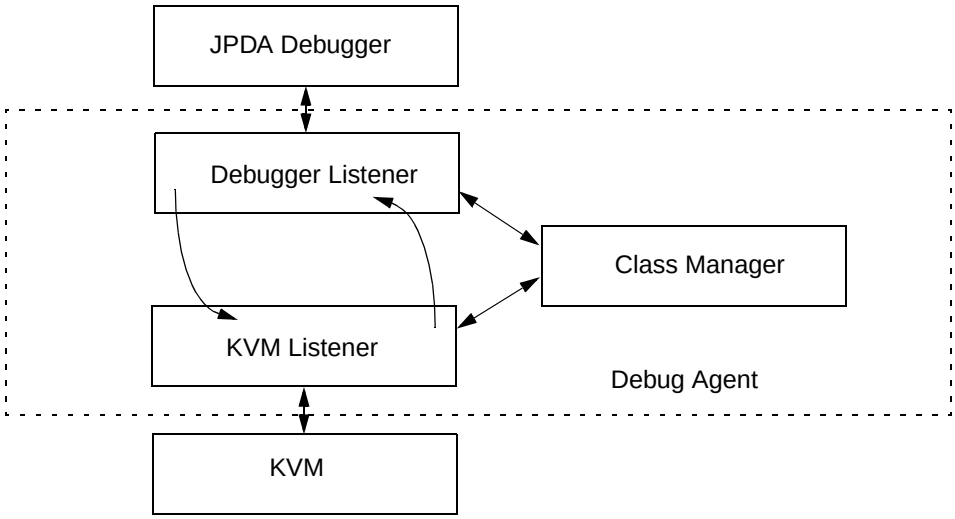

FIGURE 3 Debugger and KVM connections

In a typical scenario, the KVM is started with the `-debugger` flag, which puts it into a *debugger enabled* mode. In this mode the KVM listens on a socket for a connection from the debug agent. When the debug agent is started, it connects to

this socket, and then listens on another socket for a connection from the debugger. When the debugger connects, it issues the *JDWP handshake* command, which consists of the string "`JDWP-Handshake`". The debug agent acknowledges by reflecting this string back to the debugger. Meanwhile, the debug agent has sent the handshake command to the KVM and the KVM has responded back with information concerning which optional events it supports. The `KVMListener` then queries the KVM for a list of all the classes that are currently loaded into the VM. This information is used to build a hash table of `ClassFile` objects that is used later when the debugger requests information about a specific class (such as line number information, method information, and so forth.) At this point, each thread is listening for packets. The KVM sends a `VMInit` event to the debugger via the debug agent, which indicates to the debugger that the KVM is starting its execution of the Java application. The debugger might also send packets that indicate to the KVM to start up other events such as `ClassPrepare` or `ClassLoad`.

The communication code for the debug agent is in source file `SocketConnection.java`. In this file, each object (`KVMListener` and `DebuggerListener`) creates a thread of execution that waits for packets to arrive from its respective socket. If the packet is a command packet (the `Packet.Reply` bit is not set), then it puts that packet on a `packetQueue` list (see file `ProxyListener.java`) and a notification is sent to any object waiting on that queue. The packet is then extracted from the queue by whatever listener is waiting for that packet on that queue. In the `run` method for the `KVMListener` and `DebuggerListener`, each packet is analyzed to determine if the debug agent needs to process the packet or whether it is to be transmitted to the other object for further processing.

16.2.2 Packet processing

The `DebuggerListener` object intercepts a number of packets as is evident by examining the code for the large switch statement located after the call to `waitForPacket`. When `waitForPacket` returns with a packet, the debug agent first creates a new `PacketStream` object, then checks to see if the debug agent needs to process that packet (For example, the `SENDVERSION_CMD` packet is processed by the debug agent directly, and a response is created and sent back to the debugger without any interaction with the KVM.) A more complex command would be the `FIELDS_CMD` of the `REFERENCE_TYPE_CMDSET`. For this command, the debugger has passed in a *class id*, which is used by the debug agent to find a `ClassFile` object via the `ClassManager.classMap` object. The `classMap` object is filled by the `KVMListener` object when it receives the `ClassPrepare` events from the KVM. Once the debug agent has obtained the `ClassFile` object, it uses the `getAllFieldInfo` method to obtain a *list* of fields, and iterates through this list passing the information back to the debugger. Once again, there is no interaction with the KVM.

Similarly, within the source file for the `KVMListener.java`, the `KVMListener` object intercepts the `CLASS_PREPARE` events (events whose type is equal to the constant `JDWP_EventKind_CLASS_PREPARE`) that are passed up from the KVM. `KVMListener` creates a new `ClassFile` object via the call to

manager.findClass and inserts it into the ClassManager.classMap hashtable. KVMListener then passes the event to the debugger so that it can process the event as well.

16.3 Debugger support within KVM

The debugger support within the KVM consists primarily of four source (.c) files under the VmExtra/src directory and three header (.h) files under VmExtra/h directory. All debugger code is included with the conditional compilation flag, ENABLE_JAVA_DEBUGGER. If this flag is enabled, and the KVM is rebuilt, then the Java debugger support is included within the KVM. If Java debugger support is not desired, set this define in main.h to 0.

Note – If your target platform or port does not require Java-level debugging support, we recommend turning the debugging code off at compile time (in file main.h or in your platform-specific machine_md.h file):

```
#define ENABLE_JAVA_DEBUGGER 0
```

This will make the KVM executable much smaller.

The primary file for the Java debugger support within the KVM is the source file debugger.c. This file contains all the support needed for the KDWP API. Socket communication is handled by the code in file debuggerSocketIO.c. The debuggerInputStream.c and debuggerOutputStream.c files contain the code for handling the transmission of data being sent to/from the debugger support functions in debugger.c. The code in debugger.c file services all the KDWP requests that are sent by or through the debug agent. The function ProcessDebugCmds handles the parsing of input packets to determine which command set and what command within the command set the packet is referring to. This function then determines the appropriate function that is to be invoked for handling this command. The inputStream handle as well as the outputStream handles are passed as parameters, and used for handling the reply back to the debug agent. For performance reasons, most commands use a global inputStream and outputStream. If these are already in use, another one is allocated from the heap.

16.3.1 Events

Events are essentially commands generated by the KVM. Events are passed up to the debug agent, which may in turn pass them up to the debugger. The code for handling an event will appear as follows:

```
#if ENABLE_JAVA_DEBUGGER
{
  CEModPtr cep = GetCEModifier();
  cep->thread = thisThread;
  setEvent_ThreadStart(cep);
  FreeCEModifier(cep);
}
#endif /* ENABLE_JAVA_DEBUGGER */
```

This creates a new `CEModPtr` structure that contains state information for this particular event. It then invokes a routine in `debugger.c`, which attempts to send the event. A typical event routine in `debugger.c` first determines if the event attempting to be sent has been enabled by a previous *Set Event* command from the debugger (via the `checkNOTIFY_WANTED` macro.) Then, `findSatisfyingEvent` is invoked to determine if this particular event matches an event request sent down from the debugger. The `findSatisfyingEvent` function also checks the event counter as well as any modifiers that the debugger has applied to this event. If the event passes, then it is sent on the `outputStream`. After an event is sent, `handleSuspendPolicy` is invoked to process whatever suspend policy the debugger has attached to this event when the debugger had issued the *Set Event* command. Some events such as breakpoints or single stepping will generally have a suspend policy of `ALL`, which means that all threads are suspended and that the KVM will essentially spin through the reschedule loop at the top of the interpreter loop waiting for a thread to resume. The *Resume* command will eventually come from the debugger when the user issues a *Continue* command or when the user explicitly issues a *Resume Thread* command.

In certain situations, events need to be deferred. This is because it is not possible to send the event to the debug agent and subsequently suspend the KVM threads, since the interpreter might be in the midst of executing a byte code. Thus in such cases, `insertDebugEvent(cep)` is called instead of `setEvent_XXX(cep)`, as shown in the example above. At the top of the interpreter loop (see `VmCommon/src/execute.c`), the events are checked, and if there is a pending event, it is sent when it is safe to do so.

16.3.2 Breakpoints

When a *Set Event* command is received to add a breakpoint, the code for handling the breakpoint event determines if the opcode at that particular location is a *Fast opcode*. If so, then the original opcode must be retrieved from the *inline cache* before the breakpoint is added. The original opcode is stored in an `EVENTMODIFIER` structure that is pointed to by the `VMEvent` structure for this particular event. When the Java bytecode interpreter hits the `Breakpoint` opcode (see

bytecodes.c), and if not *single stepping*, then the handleBreakpoint function is invoked. This function restores the original opcode into the thread structure for the CurrentThread, at the point where the breakpoint had been entered. It then also sends an event to the debugger via the debug agent. Eventually, the user will press the *Continue* button on the debugger, which results in all threads to resume execution. The RESCHEDULE macro (see execute.h) includes some code in it for determining if this thread was just at a breakpoint, and if so, it will retrieve the next bytecode from a known location within the thread structure. The code within the interpreter loop will then execute this instruction.

16.3.3 Single stepping

When the debugger issues a SingleStep event request, the code in debugger.c must determine which type of step function it is (that is, *step by bytecode* or *step by line*), whether the step is a *Step Into* (step into a function), *Step Over* (step over calls to functions; that is, do not single step into another function), or *Step Out* (go back to the function that called this function). Additionally, if it is a *step by line*, then KVM needs to know what the code offset is for the next line number. To obtain this information, KVM calls a private API within the debug agent to return the target offset and the next line offset. The debug agent returns this information back to the KVM, which stores it into a stepInfo structure, which is part of the threadQueue structure (see thread.h.) Within the interpreter loop, a flag is checked to determine if this particular thread is in single step mode. If so, then the handleSingleStep function in debugger.c is invoked to process this *single step*. The handleSingleStep function determines if the instruction pointer has reached the target offset or if it has popped up a frame or if it has gone beyond the target offset. Depending on the type of stepping being performed, this function will determine when to send a SingleStep event to the debugger. In most cases, if the user is single stepping line by line, and when the code offset is equal to the target offset, it results in a SingleStep event to be sent to the debugger. All threads are typically suspended at this point, and as was the case for the breakpoint scenario above, the KVM will wait until the debugger resumes the threads via a Continue command or a subsequent SingleStep event.

16.3.4 Suspend and nosuspend options

It is desirable in certain IDE environments such as Borland's JBuilder to provide an option similar to that available in J2SE for starting up the KVM in two different debugging modes. Thus, as of KVM 1.0.3, the KVM debugger can be started in the following two modes:

```
kvm -debugger -suspend ...
kvm -debugger -nosuspend ...
```

In the *suspend* mode (this is the default), the KVM stops all Java threads upon VM startup and waits for further commands from the IDE (development and debugging environment) before the debugging session proceeds any further. In the *nosuspend* mode, the Java threads start running immediately when the KVM is started.

In the most common cases, the KVM debugger is usually invoked in the *suspend* mode. Unless the application program being debugged requires substantial processing or is recursive in nature, it may not make much sense to invoke the KVM in the *nosuspend* mode. This is because it is very likely that a simple application program may complete execution long before the debugger IDE is able to issue any commands to the KVM.

16.4 Using the Debug Agent and the JPDA Debugger

In order to run the debug agent, it is necessary to build the application class or classes being debugged to include debug information. It is also necessary to transform the application class file(s) using the preverifier. Then, after the KVM is invoked on a specified host and port, the debug agent can be started so that it listens to KVM requests on the KVM port, and a local port is specified for connecting with a JPDA-compatible debugger.

The following section summarizes the steps necessary to start a debug session in much more detail.

Note – KVM debugger functionality is integrated into the J2ME Wireless Toolkit (WTK). Therefore, if you are using the WTK, the detailed steps in the next section are not necessary.

16.4.1 Starting a debug session

To start a debug session, the following five steps are necessary:

1. **Build the application classes to be debugged with the -g option to include debug information. Then, place the output in a separate directory for transforming the resulting class file. See Chapter 13, "Class File Verification."**

   `javac -g -classpath` <*path*> `-d` <*directory*> <*class*>

   - `-g` indicates to include debug information
   - `-classpath` <*path*>, *where* <*path*> *indicates the directory in which the CLDC/KVM Java library classes and the application classes for the application being debugged are located.*
   - `-d` <*directory*>, where <*directory*> indicates the directory in which output classes will be written. The default output directory is `./output`.
   - <*class*> is the application class or classes being debugged.

2. **Invoke the preverifier for transforming the class file.**

   `preverify -classpath` <*path*> `-d .` <*directory*>

   This will transform all classes under <*directory*> and places the transformed class files in the current directory (as specified by the -d option).

3. **Start the KVM process:**

   `kvm -debugger -classpath` <*path*> `-port` <*KVM port*> <*class*>

   - `-debugger` indicates to put the KVM in debugger enabled mode
   - `-classpath` <*path*>, where <*path*> specifies the directory in which the CLDC/KVM Java library classes as well as the application classes for the application being debugged are located.
   - `-port` <*KVM port*> is the KVM port. The default KVM port is 2800. This must match the KVM port specified by the debug agent below.
   - <*class*> is the application class being debugged.

4. **Start the debug agent (debug proxy):**

   `java -classpath` <*path*> `kdp.KVMDebugProxy -l` <*localport*> `-p -r` <*KVM host*> <*KVM port*> `-cp` <*KVM_path*>

   - `-classpath` <*path*>, where <*path*> specifies the directories in which the debug proxy classes are located.
   - `-l` <*localport*>, where <*localport*> is the port that the debugger connects to.
   - `-p` indicates to use the class parser.
   - `-r` <*KVM host*>, where <*KVM host*> is the remote host name.
   - <*KVM port*> is the KVM port. As stated earlier, this port must match the KVM port specified in step 3 above.
   - `-cp` <*path*>, where <*path*> is the directory or directories where the CLDC/KVM Java library classes as well as the application classes for the application being debugged are located.

5. **Connect to the debug agent with the debugger:**

   You can use any JPDA-compliant debugger such as Forte, JBuilder or jdb. With the Forte debugger, go to the *Debug->Connect* dialog box and insert the host where the debug agent is running and the local port number that had been specified using the `-l` <*localport*> option.

Note – To download the Forte debugger and for further information on Forte, please refer to the Sun One Studio website at `http://wwws.sun.com/software/sundev/jde/index.html`. When running the Forte debugger, JDK 1.3 or later must be previously installed and be on the classpath, since only this version (or later) of the JDK includes support for the JPDA. For further information on downloading the JDK 1.3, please refer to the website at `http://java.sun.com/j2se/1.3/`.

For jdb (Java debugger), the command is as follows:

```
jdb -attach <agent hostname>:<localport>
```

16.4.2 Debugging example

If the KVM is running on a system called *sicily*, and the debug agent and debugger are running on *debughost*, then the commands for starting the debug session would appear as follows:

- On the *sicily* system, build the application test as follows:

  ```
  javac -g -classpath ../api/classes:../samples/classes
    -d output test.java
  ```

- Invoke the preverifier for building a preverified class file.

  ```
  preverify -classpath ../api/classes:../samples/classes
    -d . output
  ```

- On the *sicily* system, type the following command to invoke the KVM:

  ```
  kvm -debugger -classpath ../api/classes:../samples/classes
    -port 2800 test
  ```

- On the debughost system, assuming the current directory is `tools/kdp/classes`, then the following command would invoke the debug agent (debug proxy).

  ```
  java -classpath . kdp.KVMDebugProxy -l 1234 -p -r sicily
    2800 -cp ../../../api/classes:../../../samples/classes
  ```

- On the *sicily* system, start the JPDA-compliant debugger.

  For example, in Forte, open the source file `test.java` and insert a breakpoint at some line in the file.) In Forte, go to the `Debug->Connect` menu selection. (Newer versions of Forte have an `Attach` menu item). Then, enter `debughost` in the hostname box, and enter 1234 in the port number box. Press OK.

At this point the Forte debugger communicates with the KVM, the program starts to run, and eventually it hits the breakpoint you inserted. You can now use Forte to single-step, read or set variable values, and so forth.