

KVM Porting Guide

KVM 1.0



Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

1.0, May 19, 2000

Copyright © 2000 Sun Microsystems, Inc.

901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you at no charge a nonexclusive, nontransferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the K Virtual Machine (KVM) or J2ME CLDC Reference Implementation technology to use this document for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the document and you shall have no right to use the document for productive or commercial use.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, JDK, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Contents

- 1. About This Document** 1
- 2. Introduction to KVM** 3
- 3. Compiler Requirements** 5
- 4. Directory Structure** 7
 - 4.1 Overview 7
 - 4.2 Directory `kvm/VmCommon` 8
 - 4.3 Directory `kvm/VmExtra` 10
- 5. Required Port-Specific Files and Functions** 11
 - 5.1 File `machine_md.h` 11
 - 5.2 File `main.c` 12
 - 5.3 Runtime functions that require porting efforts 12
 - 5.4 Required C library functions 13
- 6. Compilation Flags, Definitions and Macros** 15
 - 6.1 General compilation options 15
 - 6.2 General system configuration options 16
 - 6.3 Palm-specific system configuration options 17
 - 6.4 Memory allocation settings 18

6.5	Garbage collection options	18
6.6	Interpreter execution options	19
6.7	Debugging and tracing options	20
6.7.1	Including and excluding debugging code	20
6.7.2	Tracing options	20
6.8	Networking and storage options (Generic Connections)	21
6.9	Error handling macros	21
6.10	Miscellaneous macros and options	22
7.	Virtual Machine Startup	23
7.1	Command line startup	23
7.2	Alternative startup strategies	24
7.3	Using a JAM (Java Application Manager)	24
8.	Class Loading	25
8.1	Generic interface	25
8.2	JAR file reader	26
9.	64-bit Support	29
9.1	Setup	29
9.2	Alignment issues	31
10.	Native Code	33
10.1	Native code lookup tables	33
10.2	Implementing native methods	34
10.2.1	Include files	34
10.2.2	Accessing arguments from native methods	34
10.2.3	Returning a result from a native function	35
10.2.4	Shortcuts	36
10.2.5	Callbacks	36
10.2.6	Exception handling in native code	36

10.2.7	Useful functions in native code	36
10.2.8	Garbage collection issues	37
10.2.9	Initialization and reinitialization of global variables	38
10.3	Asynchronous native methods	39
10.3.1	Design of asynchronous methods	40
10.3.2	Implementation of asynchronous methods	41
11.	Event Handling	43
11.1	High-level description	43
11.1.1	Synchronous notification (blocking)	43
11.1.2	Polling in Java code	44
11.1.3	Polling in the bytecode interpreter	44
11.1.4	Asynchronous notification	45
11.2	Parameter passing and garbage collection issues	47
11.3	Implementation in KVM	47
11.4	Battery power conservation	49
12.	Class File Verification	51
12.1	Using the new verifier	52
12.1.1	Invoking the preverifier	52
12.2	Preverifier options	52
12.3	Porting the new verifier	53
12.3.1	Compiling the preverifier	53
13.	JavaCodeCompact	55
13.1	JavaCodeCompact options	55
13.2	Porting JavaCodeCompact	56
13.3	Compiling JavaCodeCompact	57
13.4	JavaCodeCompact files	57
13.5	Executing JavaCodeCompact	58
13.6	Limitations	60

14. Java Application Manager (JAM)	61
14.1 Using the JAM to install applications	62
14.1.1 Application launching	63
14.1.2 Application updating	64
14.2 JAM components	65
14.2.1 Security requirements	65
14.2.2 JAR file	65
14.2.3 Application Descriptor File	65
14.2.4 Network communication	67
14.3 Application lifecycle management	67
14.3.1 Termination of the KVM Task	67
14.4 Error handling	68
14.4.1 Error conditions	68

Figures

- FIGURE 6-1 Error handling 22
- FIGURE 10-1 A native method 34
- FIGURE 10-2 One temporary root 38
- FIGURE 10-3 Multiple temporary roots 38
- FIGURE 10-4 Creating a global root 38
- FIGURE 10-5 Asynchronous implementation of `ReadBytes` 41
- FIGURE 10-6 Alternative asynchronous implementation of `ReadBytes` 42

Tables

TABLE 3-1	Basic types	5
TABLE 3-2	Floating Point types	5
TABLE 4-1	Distribution directories	7
TABLE 4-2	Files in VmCommon	8
TABLE 4-3	Files in VmExtra	10
TABLE 9-1	64-bit types	29
TABLE 9-2	Implementing longs	30
TABLE 9-3	Implementing both longs and floats	30
TABLE 10-1	Macros for popping arguments from the stack	35
TABLE 10-2	Macros for pushing arguments onto the stack	35
TABLE 10-3	Macros used in asynchronous methods	40

About This Document

This document provides information for porting the Sun Microsystems reference implementation of the K Virtual Machine (KVM) to a new platform.

Introduction to KVM

KVM (also known as the K Virtual Machine or as the KJava Virtual Machine) is a compact, portable Java[™] virtual machine intended 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 KVM 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 in the range 40 kilobytes to 80 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 a 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).

Further information on KVM and Java 2 Micro Edition is available in separate documents (*The K Virtual Machine (KVM)*, *A White Paper, KVM Technical Specification*, and *Connected, Limited Device Configuration Specification*, Java Community Process, Sun Microsystems, Inc.).

KVM is derived from a research system called *Spotless* developed originally at Sun Microsystems Laboratories. 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*.

Compiler Requirements

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 3-1.

TABLE 3-1 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 3-2.

TABLE 3-2 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:

- Metrowerks CodeWarrior Release 6 for Palm,
- Sun DevPro C Compiler 4.2 on Solaris,
- GNU C compiler on Solaris,
- Microsoft Visual C++ 6.0 Professional on Windows 98 and Windows NT 4.0.

The only non-ANSI feature in the source code is its use of 64-bit integer arithmetic.

1. Note that in Java, the type `long` is always 64 bits. Table 1 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.

Directory Structure



4.1 Overview

Unzip the distribution into any directory of your choice. It creates a directory with the following subdirectories:

- api
- bin
- build
- docs
- jam
- kvm
- samples
- tools

The contents of these directories are detailed in TABLE 4-1.

TABLE 4-1 Distribution directories

Subdirectory	Description
api	Contains the Java library class 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.

TABLE 4-1 Distribution directories

Subdirectory	Description
kvm	Contains the source code of KVM.
samples	Contains the source code and icons of a number of sample applications.
tools	Contains the source code and icons of a number of tools (JavaCodeCompact, preverifier, Palm tools) that are provided with this release.

4.2 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/VmPort/src/` and `kvm/VmPort/h/`, where *Port* is replaced by the name of your platform (e.g., `kvm/VmWin`, `kvm/VmPilot`, `kvm/VmLinux`.)

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

TABLE 4-2 gives an overview of the KVM source code files contained in `kvm/VmCommon/src/` and `kvm/VmCommon/h/`.

TABLE 4-2 Files in `VmCommon`

File	Description
<code>StartJVM.c</code>	Virtual machine startup and command line argument reading.
<code>cache.h</code> <code>cache.c</code>	Inline caching operations for speeding up method lookup.
<code>class.h</code> <code>class.c</code>	Runtime data structures and operations for representing Java classes.
<code>events.h</code> <code>events.c</code>	Implementation of a stream-based protocol for event handling.
<code>fields.h</code> <code>fields.c</code>	Runtime data structures and operations for representing fields of objects.
<code>frame.h</code> <code>frame.c</code>	Stack frame and exception handling operations.

TABLE 4-2 Files in VmCommon

File	Description
garbage.h garbage.c	Garbage collector and memory management.
global.h global.c	Miscellaneous global variables.
hashtable.h hashtable.c	Hashtable implementation that is used internally by the virtual machine.
interpret.h interpret.c	Bytecode interpreter.
jar.h jarint.h jartables.h jar.c	Jar file reader.
loader.h loader.c	Class loader.
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 flags and system-wide definitions.
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.c	Operations for accessing Java system properties.
rom.h	Macros needed by the ROMizer (JavaCodeCompact).
runtime.h	Definition of certain runtime functions that are commonly overridden by ports.
thread.h thread.c	Runtime data structures and operations for Java thread management and multithreading.
verifier.h verifier.c	Classfile verifier (see Chapter 12 for details).

4.3 Directory kvm/VmExtra

The directory `kvm/VmExtra/` contains additional components that are potentially useful to a large number of ports. These files include support for the “Spotlet” application model inherited from the Spotless JVM (precursor of KVM), an implementation of the most commonly needed networking protocols for Windows, and an implementation of an experimental storage protocol that uses the new Generic Connection framework.

Also the directory defines some additional macros for asynchronous event handling, and defines the low-level file operations and virtual machine startup operations needed on non-embedded target platforms such as Windows and Solaris.

A description of the `VmExtra` files is provided in TABLE 4-3.

TABLE 4-3 Files in `VmExtra`

File	Description
<code>async.h</code>	Macros for supporting asynchronous notification (see Section 10.3, “Asynchronous native methods” and Section 11.1.4, “Asynchronous notification”).
<code>loaderFile.c</code>	Low-level binding between the file system, class loader and JAR reader for those platforms that have a “real” file system.
<code>main.c</code>	Default main program for those platforms that have a file system and support VM startup from a command line.
<code>nativeSpotlet.h</code> <code>nativeSpotlet.c</code>	Low-level event handling and graphics code needed for supporting the Spotlet application model inherited from Spotless JVM (precursor of KVM). Most of the necessary event handling operations in KVM are now done using alternative mechanisms (see <code>event.c</code>).
<code>network.c</code> <code>networkPrim.h</code> <code>networkPrim.c</code>	Implementation of most commonly used network protocols for Windows.
<code>resource.c</code>	Implementation of a stream-based protocol for reading external resources.
<code>storage.h</code> <code>storage.c</code> <code>storagePrim.c</code>	Experimental implementation of a protocol for accessing storage systems in a generalized way.

Required Port-Specific Files and Functions

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

5.1 File `machine_md.h`

Every port must provide a file named `VmPort/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, 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.

5.2 File `main.c`

You will generally need to provide a new version of `main.c` that is suitable for your target platform. The default KVM 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.

5.3 Runtime functions that require porting efforts

Each port must define the functions given below. 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 **heapPtrPtr)`
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, `*heapPtrPtr` 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.
- `GetNextKVMEvent(KVMEEventType *evt, 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. See Chapter 11 for details.
- `void InitializeVM(int *argc, char **argv)`
Initialize the virtual machine in whatever way is necessary. On many of the current ports, this is a macro that does nothing.

- `void InitializeNativeCode(int *argc, char **argv)`
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 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.
- `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.

The functions `InitializeVM` and `InitializeNatives` are called, in that order, before any global variables have been set and before the memory-management system has been initialized. Each of these functions is passed a *pointer* to the `argc` and `argv` that were originally given to the `StartJVM()` function. These functions can modify the argument count and argument vector, if they so choose.

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.

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()
CallAsynchronousFunction_md()
enterSystemCriticalSection()
exitSystemCriticalSection()
```

These functions are described in §10.3.

5.4 Required C library functions

The KVM uses the following C library functions:

- String manipulation: `strcat`, `strchr`, `strcmp`, `strcpy`, `strncpy`, `strlen`
- Moving memory: `memcpy`, `memmove`, `memset`, `memcmp`
- Printing: `atoi`, `sprintf`, `fprintf`, `putchar`
- 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 only 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`, the machine-specific ports provided with this release, and the optional Java Application Manager (JAM) component will 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 PalmOS function is `MemSet(location, count, value)`.

Compilation Flags, Definitions and Macros

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

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

```
#define COMPILER_SUPPORTS_LONG 1
```

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 for more details.

6.2 General system configuration options

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

```
#define INCLUDE_ALL_CLASSES 1
```

Includes or excludes non-CLDC classes from the target system. Turning this option on also includes those components (e.g., graphics support, network protocol implementations) that are not part of the CLDC Specification.

```
#define IMPLEMENTS_FLOAT 0
```

Turns floating point support in KVM on or off. Should be off in CLDC-compliant implementations.

```
#define USES_CLASSPATH 1
```

Turns CLASSPATH support on or off. If this option is turned on, KVM uses the host system environment variable CLASSPATH to determine the location of Java class files.

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

Path separator character used in CLASSPATH. This definition is meaningful only when utilizing the USES_CLASSPATH option.

```
#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 13 for details.

```
#define USE_JAM 1
```

Includes or excludes the optional Java Application Manager (JAM) component in the virtual machine. Refer to Chapter 14 for details.

```
#define ASYNCHRONOUS_NATIVE_FUNCTIONS 0
```

Instructs the KVM to know whether asynchronous native functions are used or not. Refer to Section 10.3, “Asynchronous native methods” and Chapter 11 for details.

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 version of KVM, but they may be useful as a starting point for certain ports.

```
#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 the Palm.

```
#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 the Palm.

6.4 Memory allocation settings

The following definitions affect the amount of memory KVM allocates.

```
#define MAXIMUMHEAPSIZE 65024 /* 0xFE00 */
```

The Java heap size that KVM allocates upon virtual machine startup.

```
#define INLINECACHE_SIZE 40
```

The size of a special inline cache area that KVM reserves upon virtual machine startup if the `ENABLEFASTBYTECODES` option is turned on. The size is expressed as a number of inline cache entries (each entry requires 12-16 bytes depending on your target platform.)

```
#define STACKCHUNKSIZE 64
```

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 more space is needed.

```
#define STRINGBUFFERSIZE 512
```

The size (in bytes) of a statically allocated area that the virtual machine uses internally in certain 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 changes if the virtual machine calls host-system specific native functions (e.g., graphics functions) that perform dynamic memory allocation outside the Java heap.

6.5 Garbage collection options

The following options allow you to improve the precision of the conservative garbage collector. Read the documentation in `VmCommon/h/garbage.h` and `VmCommon/h/main.h` for further information.

```
#define SAFEGARBAGECOLLECTION 1
#define TESTNONOBJECT_DEPTH 3
```

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.

```
#define EXCESSIVE_GARBAGE_COLLECTION 0
```

Note – We are planning to implement a new garbage collector in a later version of the KVM. The options above are likely to change when the new garbage collector is available.

6.6 Interpreter execution options

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 15-20%, but increases the size of the virtual machine by several kilobytes. Note that bytecode replacement cannot be done on those target platforms in which bytecodes are stored in non-volatile memory (e.g., 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 500
```

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 11.4, “Battery power conservation” for further details.

6.7 Debugging and tracing options

KVM includes a large number of useful debugging and tracing functions and options to facilitate porting efforts. All these options should be turned off in a production release.

6.7.1 Including and excluding debugging code

```
#define INCLUDEDDEBUGCODE 0
```

Includes a large amount of debugging and logging code that is useful when porting the virtual machine to a new platform.

```
#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.

6.7.2 Tracing options

The following options allow you to turn individual execution tracing and printing options on or off. All output is printed to `stdout`. Note that many of the options require the `INCLUDEDDEBUGCODE` and/or `ENABLEPROFILING` options to be turned on.

```
#define TRACEMEMORYALLOCATION 0
#define TRACEGARBAGECOLLECTION 0
#define TRACEGARBAGECOLLECTIONVERBOSE 0
#define TRACECLASSLOADING 0
#define TRACECLASSLOADINGVERBOSE 0
#define TRACETHREADING 0
```

```
#define TRACEBYTECODES 0
#define TRACEMETHODCALLS 0
#define TRACEVERIFIER 0
#define TRACEEXCEPTIONS 0
#define TRACEEVENTS 0
#define TRACEMONITORS 0
#define TRACE_STACK_CHUNKS 0
#define TRACE_FRAMES 0
```

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

```
#define TERSE_MESSAGES 0
```

6.8 Networking and storage options (Generic Connections)

The CLDC Specification defines a new Generic Connection framework that is intended for supporting networking, storage, resource management and other related things in an efficient and extensible fashion. KVM can take advantage of some of these mechanisms internally by using the macros defined below.

```
#define GENERICEVENTS 1
```

Implements event handling inside the KVM using the generic connection mechanism. This is the default mechanism (older event handling code has been removed from the virtual machine.)

```
#define GENERICNETWORK 0
#define GENERICSTORAGE 0
```

Allows the virtual machine to take advantage of certain networking and storage capabilities that have been implemented using the generic connection mechanism.

6.9 Error handling macros

The interpreter uses code of the form shown in Figure 6-1.

If there is a call to the macro `ERROR_THROW(int)`, 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 `ERROR_THROW` jumps to the innermost `ERROR_CATCH` error handling code.

```
ERROR_TRY {  
    normal code  
} ERROR_CATCH (error) {  
    error handling code  
} ERROR_END_CATCH  
    always continue here
```

FIGURE 6-1 Error handling

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

6.10 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.

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 practices, 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

```
-debug  
-verbose  
-classpath <list of directories>
```

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 list of directories above is a single string in which the directories are separated by the `PATH_SEPARATOR` character.

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 startup strategies

If your implementation does not start the virtual machine from a command line (e.g., 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, e.g., a command line language, graphical file manager, or even a file system. To facilitate the porting of KVM to such platforms, KVM provides a reference 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 0
```

The JAM reference implementation assumes that applications are available for downloading as JAR files by using a network or storage protocol implemented using the Generic Connection framework (refer to the CLDC Specification for further details.) The JAM reads the contents of the JAR file and an associated descriptor file, 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 14, “Java Application Manager (JAM),” for further information on the JAM.

Class Loading

The KVM code includes an implementation for reading class files as files in a directory, and as entries in a compressed JAR file.

If you need to provide an alternative method for loading class files, you must define your own class loading mechanism. The default implementation in `VmExtra/src/loaderFile.c` can be used as a starting point for platform-specific implementations.

8.1 Generic interface

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 initializeClassPath()`
The code must initialize the variable `ClassPathTable` and any other variables needed for file loading. This function only needs to be defined if the preprocess constant `USES_CLASSPATH` has a non-zero value (this is the default). The value in `ClassPathTable` is a root for garbage collection, and must either be `NULL` or be an object allocated from the heap.¹

1. In a future release of KVM, this function might be replaced with the more generic `initializeFileLoading()`. The symbol `USES_CLASSPATH` then will no longer be necessary. The variable `ClassPathTable` will then appear only in generic code.

The C preprocessor constant `PATH_SEPARATOR` indicates the character that separates directories in the class path. Its default value is `':'`. Windows and other similarly based implementations need to change this value to `'\'`.

- `FILEPOINTER openClassfile(const char *className)`
Open the class file containing the class whose name is `className`. The variable `className` is a fully qualified class name that use slashes (`'/'`) as the package separator.
- `unsigned char loadByte(FILEPOINTER ClassFile)`
`unsigned short loadShort(FILEPOINTER ClassFile)`
`unsigned long loadCell(FILEPOINTER ClassFile)`
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 ClassFile, char *buffer, int len)`
Load the next `len` bytes from the class file into the indicated buffer.
- `void skipBytes(FILEPOINTER ClassFile, unsigned int len)`
Skip the next `len` bytes in the class file.
- `void closeClassfile(FILEPOINTER ClassFile)`
Close the indicated class file. Close any system resources (such as file handles or database records) associated with the class file.

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

8.2 JAR file reader

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 `loaderFile.c` and in `jar.c` for decompressing classfile entries and resources from a JAR file.

The function

```
bool_t findJARDirectories(FILE *jarFile,
                          unsigned long *localDirectory,
                          unsigned long *centralDirectory)
```

tries to find the local directory and the central directory in the already opened JAR file. If successful, it returns `TRUE` and sets the variables `localDirectory` and `centralDirectory` to the appropriate values. If it is unable to find these directories, it returns `FALSE`; in this case, the opened file is almost certainly not a jar file.

The function

```
JAR_DataStreamPtr
loadJARfile(const char *jarFileName,
            unsigned long localDirectory,
            unsigned long centralDirectory,
            const char *fileName)
```

returns a data structure containing the decompressed contents of the specified `fileName`. It returns `NULL` if the file name is not found in the Jar file, or if there was some problem extracting it.

`loadJARfile` makes use of a function `inflate()`. This function can be used to decompress anything (including zip file entries) that has been compressed using the “deflate” algorithm.

Normally, `inflate()` takes the arguments:

```
bool_t inflate(FILE *compressedData,
               int   compressedLength,
               unsigned char *decompressedData,
               int   decompressedLength)
```

- `compressedData`:
a `FILE` that is positioned to read the first byte of compressed data
- `compressedLength`:
length of the compressed data
- `decompressedData`:
a buffer into which to write the result
- `decompressedLength`:
the length of the decompressed result

Note that both `compressedLength` and `decompressedLength` must be *exact*. The `inflate` algorithm considers it an error if the compressed data isn't exactly `compressedLength` bytes long, or if the decompressed data isn't exactly `decompressedLength` bytes long.

`inflate()` returns `TRUE` if it is successful in inflating the data, and `FALSE` if it finds any error. The position of `compressedData` in the underlying file, and the contents of `decompressedData` are undefined if `inflate()` returns `FALSE`.

The `inflate()` algorithm may allocate memory from the heap. It is the caller's responsibility to ensure that the `decompressedData` array, if it has been allocated from the Java heap, is protected from garbage collection. By setting the flag

```
#define JAR_INFLATER_USES_STDIO 0
```

the first argument to `inflate()` is instead

```
bool_t inflate(unsigned char *compressedData, . . . .)
```

In this case, you must pass an array containing the bytes to be decompressed as the first argument.

This option is useful for those ports that do not support I/O, and for which downloaded applications are stored in memory. In this case, the caller must ensure that the `compressedData` array is protected from garbage collection.

64-bit Support

If your platform supports floating-point arithmetic, your compiler must provide appropriate support.

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

9.1 Setup

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 9-1.

TABLE 9-1 64-bit types

Type	Description
<code>long64</code>	A signed 64-bit integer
<code>ulong64</code>	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, you would write:

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

Your compiler does not support 64-bit integers¹: 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.

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 9-2. If your platform supports floating point, you must also define the functions shown in Table 9-3.

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

TABLE 9-2 Implementing longs

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

TABLE 9-3 Implementing both longs and floats

Function or Constant	Java equivalent
<code>long64 float2ll(float f);</code>	<code>(long)f</code>
<code>long64 double2ll(double d);</code>	<code>(long)d</code>
<code>float ll2float(long64 a);</code>	<code>(float)a</code>
<code>double ll2double(long64 a);</code>	<code>(double)a</code>

1. Or your code must be strictly ANSI standard.

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

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 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
```

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

```
#define NEED_DOUBLE_ALIGNMENT 1
```

The compiler can generate better code when these values are 0.

Native Code

KVM does not support the Java Native Interface (JNI). Rather, the native code to be called from the virtual machine must be linked directly into the virtual machine, and must be called using the mechanisms described in this section.

Information for writing your own native functions for KVM is provided in Section 10.2, “Implementing native methods.”

10.1 Native code lookup tables

As part of the build process, you must build the lookup tables that map methods to the corresponding native implementation.

The `JavaCodeCompact` generates these tables automatically¹. 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 13. The specific details for creating the file containing the lookup tables can be found in §13.5.

The name of the C function that implements a native method must be the same name that JNI² would assign to the native method.

1. Earlier versions required the porter to build these tables by hand. This is no longer required. However it is important, now, that the C functions be given the name that JNI would give them.

2. 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>.

10.2 Implementing native methods

WARNING: You should not write 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!

The KVM reference implementation does not use JNI for native method calls. Native methods must be written extremely carefully. Inattention to detail will cause fatal errors in the virtual machine.

10.2.1 Include files

Your code containing 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.

10.2.2 Accessing arguments from 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. Figure 10-1 shows an example of this coding style:

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);  
}
```

FIGURE 10-1 A native method

An instance method (non-static method) must pop the `this` argument off the stack after it has popped the rest of the arguments. *Failing to pop the `this` argument in a native instance method will almost surely cause a fatal error in the virtual machine.*

Table 10-1 shows the macros that should be used to pop arguments off the stack:

TABLE 10-1 Macros for popping arguments from the stack

C type	Macro for popping
char, byte, int, long	<code>popStack()</code>
float	<code>popStackAsType(float)</code>
long64, ulong64	<code>popLong()</code>
double	<code>popDouble()</code>
<i>pointerType</i>	<code>popStackAsType(pointerType)</code>

In earlier versions of KVM, your code would get pointer types off the stack by calling `popStack()`, and then casting the result to the appropriate type. For example:

```
STRING_INSTANCE string = (STRING_INSTANCE)popStack()
```

This coding style should no longer be used. Instead, you should write:

```
STRING_INSTANCE string = popStackAsType(STRING_INSTANCE)
```

10.2.3 Returning a result from a 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 10-2 to push the result back onto the stack:

TABLE 10-2 Macros for pushing arguments onto the stack

C type	Macro for pushing
char, byte, int, long	<code>pushStack()</code>
float	<code>pushStackAsType(float)</code>
long64, ulong64	<code>pushLong()</code>
double	<code>pushDouble()</code>
<i>pointerType</i>	<code>pushStackAsType(pointerType)</code>

In earlier versions of KVM, you would push a pointer type onto the stack by coercing it to a `long`, and then pushing that value onto the stack. This practice is discouraged.

10.2.4 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.

In addition, you should no longer coerce the value of `topStack` to a pointer type, nor should you coerce a pointer to `long` and assign it to `topStack`. Instead, you should use the macro `topStackAsType(pointerType)`. This macro can be used both as a value and as the target of an assignment (an lvalue). This macro should also be used for accessing and setting `float` values at the top of the stack.

10.2.5 Callbacks

Native code cannot call back into Java. 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.

10.2.6 Exception handling in native code

If the native code needs to throw an error or exception, it should call the function `raiseException(string)` where the `string` argument contains the fully-qualified name (with `'/'` as the package separator) of the exception class or error class.

10.2.7 Useful functions in native code

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

- `void fatalError(string)`
The code calls this method to indicate that a serious error has occurred. The `string` 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`.
- `INSTANCE instantiateString(const char* string)`
This method converts the given C string into a Java String.
- `char *getStringContents(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(CLASS class)`
Creates a new Java instance of the specified class.
- `ARRAY instantiateArray(ARRAY_CLASS arrayType, long length)`
Creates a Java array of the specified type and length.
- `ARRAY createCharArray(const char* string)`
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 10.2.8, “Garbage collection issues”) to prevent the memory block from being garbage-collected.

10.2.8 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. As a general guideline, whenever you create a new Java object in native code, *you must make the pointer to the object visible to the virtual machine* before doing any subsequent operations that might cause the garbage collector to be called. You have several options:

- You can push the object onto the Java stack (using `pushStack`) before performing the operation. You must remember to pop it off the stack afterwards.
- You can create code of the form shown in Figure 10-2 or Figure 10-3.
- If your code initializes a C variable to point to a Java object at startup, and the contents of that variable should never be garbage collected, you can use the code shown in Figure 10-4. However, note that there is currently no function for removing a variable from the set of global roots.

```

START_TEMPORARY_ROOT(var)
    ;; var will not be garbage collected
    code that might garbage collect.
END_TEMPORARY_ROOT

```

FIGURE 10-2 One temporary root

```

START_TEMPORARY_ROOTS
    MAKE_TEMPORARY_ROOT(var1);
    ;; var1 will not be garbage collected
    code that might garbage collect
    MAKE_TEMPORARY_ROOT(var2);
    ;; neither var1 nor var2 will be garbage collected
    more code
END_TEMPORARY_ROOTS

```

FIGURE 10-3 Multiple temporary roots

```

variable = <value>
MakeGlobalRoot((cell **)value);

```

FIGURE 10-4 Creating a global root

Important: If your code needs to allocate two objects in a sequence, it is important to protect the first object before allocating the next. Otherwise, a fatal error is likely to occur.

Useful hint for debugging native functions: The garbage collector has a special mode/flag `EXCESSIVE_GARBAGE_COLLECTION` that will cause it to garbage collect before every memory allocation operation. Turning this mode on is an extremely effective technique for finding variables whose value you have forgotten to protect. For performance reasons, you should always remember to turn this mode off on a production release.

10.2.9 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 `initializeNativeMethods()` 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.

10.3 Asynchronous native methods

From the operating system viewpoint, KVM is just one process (C program) with only one 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. For this reason, KVM includes an implementation of “asynchronous native methods” described below.

10.3.1 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.

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_METHODS 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 the 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 10-3.

TABLE 10-3 Macros used in asynchronous methods

Native function macro	Asynchronous native function macro
<code>popStack</code>	<code>ASYNC_popStack</code>
<code>pushStack</code>	<code>ASYNC_pushStack</code>
<code>popLong</code>	<code>ASYNC_popLong</code>
<code>pushLong</code>	<code>ASYNC_pushLong</code>
<code>popStackAsType</code>	<code>ASYNC_popStackAsType</code>
<code>pushStackAsType</code>	<code>ASYNC_pushStackAsType</code>
<code>raiseException</code>	<code>ASYNC_raiseException</code>
<code>topStack</code>	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 10-3 have been designed so that if the symbol `ASYNCHRONOUS_NATIVE_METHODS` is 0, the asynchronous method compiles into a normal native method.

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 CallAsynchronousFunction_md(THREAD, void(f*)(THREAD))`
Call the function `f`, passing it the thread argument as its single argument. The function `f` should be called in an asynchronous manner, such as in a separate task.
- `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.

10.3.2 Implementation of asynchronous methods

We envision two possible implementations of asynchronous methods.

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

Figure 10-5 below shows one possible implementation of a method
`int readBytes(byte[] dst, int offset, int length)`
using this style of asynchronous native methods.

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

FIGURE 10-5 Asynchronous implementation of `ReadBytes`

In an alternative implementation, `CallAsynchronousFunction_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.

```

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();
}

```

FIGURE 10-6 Alternative asynchronous implementation of ReadBytes

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.

Refer to Section 11.1.4, “Asynchronous notification,” for further information on writing asynchronous code.

Event Handling

11.1 High-level description

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

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

11.1.1 Synchronous notification (blocking)

By synchronous notification we refer to a situation in which the KVM performs event handling by calling a native 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 like 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.

11.1.2 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.

11.1.3 Polling in the bytecode interpreter

The third approach to implement event handling is to use the bytecode interpreter periodically make calls to native event handling operations. This approach is a variation of the synchronous notification approach described above. This approach has been used extensively in the KVM, e.g., 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.

11.1.4 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 recently.

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 have been 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 where 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 virtual machine designer 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.

11.2 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 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 have been discussed in Chapter 10.

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 `startLongIOActivity()` and `endLongIOActivity()` 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 `startLongIOActivity()`, 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 `startLongIOActivity()` after popping the parameter from the stack the array could be garbage collected.

11.3 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:

```
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

```
GetNextKVMEvent(KVMEventType *evt, bool_t forever,  
                ulong64 waitUntil)
```

If no event occurs, the function must return `FALSE`. If an event does occur, the `evt` argument is filled with the details of the event, and the function returns `TRUE`.

The other arguments 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 have been 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 `GetNextKVMEvent` routine can then put the device “to sleep” until the next event occurs. Battery conservation issues have been discussed in more detail in the next subsection.

11.4 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

```
GetNextKVMEvent(KVMEventType *evt, 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.

Class File Verification

The existing JDK class file verifier is not suitable for small, resource-constrained devices. The JDK verifier takes a minimum of 50K binary code space, and at least 30K-100K 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 class file verifier that is significantly smaller than the existing JDK verifier. The new verifier takes about 10 kilobytes of Intel x86 binary code and less than 100 bytes of dynamic RAM at run time for typical class files. The verifier performs only 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 byte code verifier requires all subroutines to be inlined (so that there are no `jsr` and `ret` instructions) and class files to contain a `StackMap` attribute. We ship a post-javac class file transformation tool, called the *preverifier*, that inserts this attribute into normal class files. A transformed class file is still a valid J2SE class file, with an additional attribute that allows verification to be carried out efficiently at run time.

The preverifier 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` attribute. The program compiles and runs on Windows and Solaris and can be easily ported to other development platforms.

12.1 Using the new verifier

12.1.1 Invoking the preverifier

The preverification phase is usually performed at application development time on a development workstation. The preverifier is used as follows. If, for example, you compiled `Foo.java` using `javac` before:

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

Now you need to place the output of `javac` in a separate directory and then transform the resulting class files using the preverifier:

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

The above preverifier command transforms all class files under `tmpdir/` 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.

12.2 Preverifier options

The preverifier accepts a number of arguments and options.

`-classpath <directories>`

- Directories from which classes will be loaded. The directory separator is platform-specific. On Solaris a colon is used. On Win32 a semicolon is used.

`-d <directory>`

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

`@<filename>`

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

Command line options are followed by a list of class names and directory names. For each class name, the preverifier searches for a matching class file in the classpath directories and transforms the class file. For each directory name, the preverifier recursively transforms every class file under that directory.

In the future, `javac` may be changed so that it no longer generates subroutines and generates the appropriate attributes. In that case the preverifier tool will no longer be necessary.

12.3 Porting the new verifier

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

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.

12.3.1 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 loading the workspace file under 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/{Debug,Release}` directories.

JavaCodeCompact

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.

13.1 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 outputfilename`

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 “quickenized” 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. If you are using `JavaCodeCompact` for the PalmOS, you must specify `PALM` as the architecture; otherwise, you must specify `KVM` as the architecture.

13.2 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.

13.3 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 this file to indicate the location of your `javac` compiler.

13.4 JavaCodeCompact files

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

```
unix
windows
palm
```

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 §10.1). 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.

On the Palm, several files are created:

```
nativeFunctionTablePalm.c
nativeRelocationPalm.c
kvm/VmPilot/build/bin/PalmROM1001.bin
...
kvm/VmPilot/build/bin/PalmROM1010.bin
```

The file `nativeFunctionTable.c` again contains tables necessary for using native functions. It should be compiled and linked into KVM whether or not you are planning to use the other features of the `JavaCodeCompact` utility. The file `nativeRelocationPalm.c` contains relocation information needed to execute native methods. The directory `kvm/VmPilot/build/bin` contains a set of Palm resource files, that must be included in your `kvm.prc` file.

13.5 Executing JavaCodeCompact

The `JavaCodeCompact` utility is used to build 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 using `JavaCodeCompact`, 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 §13.3 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.
- 4a.[Not Palm] 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.

4b.[Palm] You should instead execute the following two commands:

```
env CLASSPATH=classes \
  JavaCodeCompact -nq -arch Palm \
  -o ROMjavaPalm.c classesPalm.zip
env CLASSPATH=classes \
  JavaCodeCompact -nq -arch Palm \
  -imageAttribute relocating
  -o nativeRelocationPalm.c classesPalm.zip
```

The file `nativeRelocationPalm.c` is included as a source file in your build. The file should be compiled and executed as follows:

```
cc -I../kvm/VmPilot/h -I../ikvmvm/VmCommon/h \
  -DRELOCATABLE_ROM -DROMIZING ROMjavaPalm.c \
  -o ROMjavaPalm
```

The resulting executable `ROMjavaPalm` is executed as follows:

```
ROMjavaPalm ../../kvm/VmPilot/build/bin/PalmROM
```

creates the resource files in the indicated directory.

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 `USING_ROMIZER` is set to a non-zero integer value. You must also ensure that the file `ROMjavaPlatform.c` (non Palm) or `nativeRelocationPalm.c` (Palm) 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.

13.6 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.

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.

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

```
#define USE_JAM 1
```

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.

Note – Currently a JAM implementation is available only for the Windows and Solaris versions of KVM.

14.1 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 providers 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 if the Java application the user selected can be installed successfully on the device before it tries to download it. This saves the user the cost of moving the Java application to the device if it cannot be installed. The file is small (several hundred bytes), while a Java application is 10-20 kilobytes, so it is much cheaper to download than the whole 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 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 room enough on the device to save it.
5. If there is room enough 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 where the JAR file was downloaded from,
- 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 just installed application selected for execution.

However, if the Use-Once tag is set to yes, JAM doesn't 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.

14.1.1 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.) In the reference implementation the user is shown the list of installed Java applications on the device, with the just-installed application selected or highlighted for execution.
2. The user selects the Java application they wish to launch (the user interface design is left up to the manufacturer).
3. The JAM executes 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.

14.1.2 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.

14.2 JAM components

14.2.1 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.

14.2.2 JAR file

JAR files are a standard feature of Java 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.

14.2.3 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-1.1” 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 (see step 6 on section 1.1.1, and section 1.3 on application updating).

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)
```

14.2.4 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 the both URLs.

14.3 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.

We use the term task 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.

We do not specify the API functions for controlling the lifecycle of the KVM, as the mechanism is vastly different from platform to platform. Instead, we require all JAM implementations to 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.

14.3.1 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.

14.4 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.

14.4.1 Error conditions

The following are a set of possible error conditions and sample messages (in English) 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 into 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 whose full URL matches exactly one 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.

