CDC Porting Guide

for the Sun Java Connected Device Configuration Application Management System

Version 1.0

Sun Microsystems, Inc.
www.sun.com

November 2005

Contents

Part VI   AMS Layer

Figures

Tables

Preface

This guide describes how to port the CDC HotSpot Implementation Java virtual machine and class library to a target platform.

The companion documents *CDC Build System Guide* and *CDC Runtime Guide* describes how to build and run a CDC Java runtime environment, including the build-time and runtime options that control functionality, testing and performance features. This guide focuses on how to use those features at runtime.

Who Should Read This Document

The primary reader is a software engineer responsible for performing the initial port of the CDC HotSpot Implementation Java virtual machine and Foundation Profile to a target platform.

The reader should also be familiar with the following topics:

- Operating systems and device drivers
- Object-oriented programming
- Java virtual machine semantics
- Java programming
- C programming
- Assembly language programming
- Open source software development tools

The porting layers of the CDC HotSpot Implementation Java virtual machine have been designed around a narrow set of porting interfaces that focus effort on the target platform rather than VM internals. So while a general understanding of compiler architecture is helpful, a deep understanding is not required. Experience with the target CPU's assembly language is necessary.

How This Book Is Organized

This document is divided into several major sections described below.

Part I: Getting Started introduces CDC HotSpot Implementation technology and outlines the planning stage for beginning a port.

Part II: HPI Layer describes the Host Programming Interface (HPI) layer, which provides the CPU and OS-specific interfaces for both the virtual machine runtime and the Foundation Profile class library.

Part III: Dynamic Compiler Layer describes the porting interfaces for the dynamic compiler. This *optional* porting layer provides a adaptng the dynamic compiler for CPUs that are not currently supported by one of the ports available from Java Partner Engineering (`http://www.sun.com/software/jpe`).

Part IV: Garbage Collector Layer describes how to build a garbage collector plug-in. These advanced APIs are *optional* since the default GC algorithms in CDC HotSpot Implementation Java virtual machine work well for a broad range of devices and applications.

Part V: GUI Layer describes the AWT porting layer for Personal Basis Profile and Personal Profile.

Part VI: AMS Layer describes the porting interfaces for CDC Application Management System.

Part VII: Appendices contains useful programming material for related porting issues like `gdb`-based debugging and programming techniques that promote system robustness under tight C stack situations.

Typographic Conventions

TABLE P-1   Typographic Conventions

Typeface	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
AaBbCc123	What you type, when contrasted with on-screen computer output	`% `**`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

TABLE P-2   Related Documentation

Topic	Title
white paper	The white paper *CDC: Java Platform Technology for Connected Devices* introduces CDC technology, standards, devices, applications and tools.
runtime options	The companion document *CDC Runtime Guide* provides runtime-oriented information for developers and testers.
build system	The companion document *CDC Build System Guide* describes the CDC build system installation, configuration and testing.
TCK	User documentation for running the TCK validation suites. • *CDC Technology Compatibility Kit version 1.0 User's Guide* • *Foundation Profile Technology Compatibility Kit version 1.0 User's Guide*

TABLE P-2 Related Documentation

Topic	Title
Java virtual machine	*Java Virtual Machine Specification, Second Edition* (Addison-Wesley, 1999) defines the Java class format and the virtual machine semantics for class loading, which are the basis for the operation of the Java runtime environment and its ability to execute Java application software on a variety of different target platforms. See http://java.sun.com/docs/books/vmspec.
Java Native Interface (JNI)	*Java Native Interface: Programmer's Guide and Specification* (Addison-Wesley, 1999) by Sheng Liang describes the native method interface used by the CDC HotSpot Implementation Java virtual machine. http://java.sun.com/docs/books/jni.
Java Virtual Machine Debugger Interface (JVMDI)	Defines an interface that allows debugger tools like jdb and third-party debuggers to interact with a debugger-capable Java runtime environment. See http://java.sun.com/products/jpda/doc/jvmdi-spec.html.
Java Virtual Machine Profiler Interface (JVMPI)	Defines an interface that allows the hprof profiler to interact with a Java runtime environment to measure application behavior. See http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.
security	*Inside Java 2 Platform Security* (Addison-Wesley, 2003) describes the Java security framework, including security architecture, deployment and customization. See http://java.sun.com/docs/books/security.
POSIX Threads	*Pthreads Programming: A POSIX Standard for Better. Multiprocessing* (O'Reilly & Associates, 1996) by Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell is an introduction to POSIX thread programming. See http://www.oreilly.com/catalog/pthread.
POSIX	*POSIX Programmer's Guide* (O'Reilly & Associates, 1991) by Donald Lewine is an introduction to the POSIX interface. See http://www.oreilly.com/catalog/posix.
ANSI Standard C Library	*The Standard C Library* (Prentice-Hall, 1991) by P. J. Plauger is a comprehensive description of the ANSI C Library. See http://www.prenhall.com/books/ptr_0131315099.html.
Berkeley Sockets	The Open Group maintains the *Single UNIX Specification* which defines standard UNIX interfaces. See http://www.unix-systems.org/version3.

TABLE P-2   Related Documentation

Topic	Title
glibc	The glibc library contains all the ANSI Standard C Library and POSIX library functions needed by the CDC HotSpot Implementation Java virtual machine. See http://www.gnu.org/software/libc/manual.
Linux	The sample implementation is based on a Linux 2.2 platform. See http://www.kernel.org.
SoftFloat Library	The *SoftFloat Library* is a software implementation of the IEEE Standard for Binary Floating-point Arithmetic. See http://www.jhauser.us/arithmetic/SoftFloat.html.
gcc Compiler	*Using and Porting the GNU Compiler Collection* (Version 2.95, 1999). See http://gcc.gnu.org/onlinedocs. See also the ARM-specific patch described in the *CDC Build System Guide*.
gdb Debugger	*Debugging with GDB* (Version 5.1.1, 2002). See http://www.gnu.org/software/gdb/documentation.
ARM Processor	*ARM System-on-Chip Architecture* (Addison-Wesley, 2000) is an introduction to the ARM processor architecture.

Accessing Sun Documentation Online

Sun provides online documentation resources for developers and licensees.

TABLE P-3   Sun Documentation

URL	Description
http://docs.sun.com	Sun product documentation
http://java.sun.com/j2me/docs	Java ME technical documentation
http://www.sun.com/software/jpe	Java Partner Engineering

Terminology

These terms related to the Java™ platform and Java™ technology are used throughout this manual.

Java technology level	(Java level)
Java technology based	(Java based)
class contained in a Java class file	(Java class)
Java programming language profiler	(Java profiler)
Java programming language debugger	(Java debugger)
thread in a Java virtual machine representing a Java programming language thread	(Java thread)
stack used by a Java thread	(Java thread stack)
application based on Java technology	(Java application)
source code written in the Java preogramming language	(Java source code)
object based on Java technology	(Java object)
method in an object based on Java technology	(Java method)
field in an object based on Java technology	(Java field)
a named collection of method definitions and constant values based on Java technology	(Java interface)
a group of types based on Java technology	(Java package)

an organized collection of packages and types basedon Java technology	(Java namespace)
constructor method in an object based on Java technology	(Java constructor)
exception based on Java technology	(Java exception)
an application programming interface (API) based on Java technology	(Java API)
a service providers interface (SPI) based on Java technology	(Java API)
developer tool based on Java technology	(Java developer tool)
system property in a Java runtime environment	(Java system property)
security framework for the Java platform	(Java security framework)
security architecture of the Java platform	(Java security architecture)

Feedback

Sun welcomes your comments and suggestions on CDC technology. The best way to contact the development team is through the following e-mail alias:

cdc-hotspot-comments@java.sun.com

You can send comments and suggestions regarding this document by sending email to: docs@java.sun.com

PART I   Getting Started

This part introduces the Connected Device Configuration HotSpot Implementation (CDC-HI) Java virtual machine and outlines the planning procedures and early stages of porting the CDC HotSpot Implementation Java virtual machine to a target platform.

This part contains the chapters:

- Introduction
- Planning

Introduction

The Connected Device Configuration (CDC) is a group of Java ME technologies for a broad range of consumer and embedded products. This porting guide describes how to port the CDC HotSpot Implementation Java virtual machine and Foundation Profile class library to a new target platform. This chapter describes the CDC HotSpot Implementation Java virtual machine, its benefits, target applications and supported platforms. Chapter 2 provides an overview of the porting process and the chapters that follow describe the different stages of a port.

1.1   CDC Technology

CDC is a Java ME technology designed to leverage Java SE technology for non-desktop systems. This includes a Java class library that is derived from the J2SE 1.4.2 Java class library by removing certain server-oriented packages and deprecated methods and adding a few CLDC compatibility classes. The result is a set of APIs that are familiar to the millions of Java SE developers yet appropriate to the needs of non-desktop devices.

The CDC HotSpot Implementation Java virtual machine uses the same external interfaces as the Java SE HotSpot Java virtual machine. Both implementations fully adhere to the *Java Virtual Machine Specification, Second Edition* (Addison-Wesley, 1999) and both support VM-level interfaces like Java Native Interface (JNI), Java Virtual Machine Debugger Interface (JVMDI) and Java Virtual Machine Profiler Interface (JVMPI). However, the design and implementation of the CDC HotSpot Implementation Java virtual machine is quite different.

One of the most important benefits of this design is portability, which is the subject of this guide. For a target platform with interface libraries based on common operating system abstractions like POSIX, the porting process should take a matter of weeks instead of months. Since many common CPUs and operating systems are already supported, much of the work is already done in the form of sample ports or

porting utility libraries. In short, the portability interface of the CDC HotSpot Implementation Java virtual machine focuses developer effort on the target platform rather than the Java virtual machine implementation.

1.2 Benefits

CDC HotSpot Implementation is a fully compliant Java virtual machine that is highly optimized for resource-constrained devices like consumer products and embedded devices. CDC HotSpot Implementation combines excellent performance and reliability with a low memory footprint to meet the needs of a broad range of product scenarios.

CDC HotSpot Implementation was designed to inhabit the world of PDAs, set-top boxes and other consumer products and embedded devices. It complies with the same Java virtual machine specification as the Java SE application environment, but its implementation is tailored to the needs of resource-constrained devices. Because product designs vary, CDC HotSpot Implementation allows device-friendly tradeoffs between performance and constrained resources. CDC HotSpot Implementation achieves best-of-class performance with a modern dynamic compiler and solid reliability for multi-threaded and low-memory conditions. In addition, CDC HotSpot Implementation's portability interfaces enable rapid modification to support new target CPUs and operating systems while maintaining excellent performance.

- *Device Support*
  - Excellent performance
  - Low memory footprint
  - Reliability

      Low-memory conditions

      Multi-threaded scenarios

  - Device friendliness

      Portable

      Configurable

- *Java Virtual Machine features*
  - Floating point
  - Multiple user-defined class loaders
  - Serialization
  - Reflection
  - Weak references
  - Full I/O and networking
  - Core features and programming model of Java SE

- *Retargetable*
  - Modular implementation

    narrow porting interfaces

    written in ANSI C and assembly language
  - CPU implementations

    ARM

    PowerPC

    MIPS

    SPARC

    x86 (interpreter only)
  - Operating systems

    Linux

    Solaris

    Darwin

    Win32
- *Design Features*
  - Dynamic compiler (JIT - *Optimized Implementation only*)

    Space efficient

    Fast

    Reliable

    Portable

    Configurable
  - Interpreter

    Fast

    Written in ANSI C

    Uses GCC extensions when available
  - Java class preloading

    Space saving

    Data sharing

    In-place execution from ROM

    Pre-loading improves startup time and avoids fragmentation
  - Runtime

    Fast startup and shutdown

No resource leaks

Small class footprint

Process model independent

Virtualized JVM state

No global variables or statics

- *Memory management*
  - Heap management

    Virtual memory not required

    Fully compactible heaps

    No fragmentation

  - Pluggable garbage collector (*Optimized Implementation only*)
  - Default generational collector

    Short pauses

    Sequential heaps

    Coexists with native system

- *Thread support*
  - Fast locking
  - Scalable and robust in heavily threaded scenarios
  - C stack safety for tight memory conditions
  - Porting layer supports both native and user-level threads

- *Standard Java VM interfaces*
  - Java SE policy-based security model
  - JNI support
  - JVMDI debug support
  - JVMPI profiling support

Planning

The CDC HotSpot Implementation Java virtual machine has been designed for easy portability to alternate target platforms. This porting guide describes the basic stages of porting the CDC HotSpot Implementation Java virtual machine: bringing up the VM runtime and integrating the Foundation Profile class library. More advanced stages like porting the dynamic compiler and implementing an alternate garbage collection algorithm are discussed in later chapters.

The porting techniques described in this guide are based on the Linux/ARM sample implementation. TABLE 3-5 and TABLE 3-6 describes implementations for other CPUs and operating systems available through Java Partner Engineering (`http://www.sun.com/software/jpe`). If your target device uses one of these CPUs or operating systems, then the porting tasks can be simplified by using different portions of these ports. In addition, the source release includes porting libraries that streamline support for most platforms.

This chapter cover the following topics:

- Target Platform Requirements
- Porting Steps
- Source Code Organization

2.1 Target Platform Requirements

The target platform requirements of the CDC HotSpot Implementation Java virtual machine are organized into the CPU and the operating system layers.

2.1.1 CPU

Note – Most of these CPU features are relavent only to a JIT port that is part of the Optimized Implementation.

The porting strategy for the CDC HotSpot Implementation Java virtual machine is based on the simplifying assumption of targeting RISC CPUs that are common in the embedded and device marketplace. Most RISC CPUs are similar and their differences are mostly limited to their registers, instruction encodings and calling conventions.

Note – Though the x86 is the dominant non-RISC CPU, the CDC-HI Java virtual machine supports the x86 only within the interpreter and not within the dynamic compiler. Supporting the x86 within the dynamic compiler would lose many of the performance efficiencies gained by concentrating on RISC architectures.

The following list identifies some of the RISC features important to the CDC-HI Java virtual machine JIT porting interface.

- *RISC-like architecture*
    - 32-bit integer and address size.
    - 2's complement integer representation.
    - Uniform 1-word instruction length.
    - Lots of registers.
    - Uniform set of general-purpose registers (except floating point) available for integer arithmetic and pointers.
    - Some parameters are passed in registers.
    - Return value appears in a register.
    - Load-store architecture. ALU instructions operate only on registers and immediates. Values must be explicitly moved between the ALU and memory with load and store instructions.
    - Interlocked load/store instructions to support fast locking.
    - Byte-addressable memory.
    - Allocating large contiguous memory regions for the code buffer and heap.
    - "call far" instruction that can reach the size of the code buffer.
    - Register-relative memory addresses for load and store instructions.
- *Dynamic compiler-specific requirements*
    - Ability to flush the i-cache.
    - Write instructions as data into memory.
    - Execute instructions out of memory.
- *Non-assumptions*
    - Byte order within a word.
    - Direction of C stack growth.
    - Presence or absence of delay slots following branches.

- Any particular arrangement of dedicated registers, or those available for use by generated code.

The CDC HotSpot Implementation Java virtual machine has been ported to several target CPUs. See TABLE 3-5 for a list of supported CPU ports available through Java Partner Engineering (`http://www.sun.com/software/jpe`).

Note – The companion document *CDC Build System Guide* describes the cross-compiler and other developer tools used by the CDC build system.

2.1.2 Operating System

The operating system portability interfaces for the CDC HotSpot Implementation Java virtual machine are based on common operating system abstractions like POSIX and ANSI I/O. If these interfaces are available on the target platform, then the porting process is mostly a direct mapping from the HPI to the platform's system interfaces. Chapter 3 describes the standard system interfaces required by the HPI:

- Memory management.
  - Uniform address space (not segmented).
  - UNIX-like memory allocation functions.
- ANSI Standard I/O.
- POSIX thread management.
  - Easily ported to a POSIX thread library.
  - Alternate fast locking implementations available.
  - Don't need separate processes.
  - Don't need a large C stack per thread.
  - Monitor based.
- Berkeley sockets.

The CDC HotSpot Implementation Java virtual machine has been ported to several target operating systems. These include Linux and several other UNIX-like operating systems. See TABLE 3-6 for a list of supported operating system ports available through Java Partner Engineering (`http://www.sun.com/software/jpe`).

The implementation requirements for the system library layer is largely shaped by the Java Native Interface (JNI) which provides a standard mechanism for Java classes to execute native methods. JNI provides a calling convention that allows a Java virtual machine to execute native methods on a target platform

2.2 Porting Steps

The porting layers of the CDC HotSpot Implementation Java virtual machine were designed to make porting straightforward and to focus effort on issues that affect performance. This is achieved through simplifying assumptions like common OS and CPU abstractions as well as using example ports and porting utility libraries.

The steps described below will help introduce the CDC HotSpot Implementation technology and prepare the way for a successful port.

1. **Start by working with a binary reference implementation as described in the** *CDC Runtime Guide***.** This will introduce the mechanics of using the Java application launcher and even more advanced topics like tuning the runtime performance of the dynamic compiler.

2. **Work with the reference build environment as described in the** *CDC Build System Guide***.** This will introduce the mechanics of performing builds and testing the CDC Java runtime environment.

3. **Perform a basic port that supports interpreter-only operation.** In most cases, the common OS and CPU abstractions make this a straightforward task. There are some basic assembly language glue routines to write. Start by reading the header files in `src/share/javavm/include/porting`, which have many descriptive comments that are not duplicated in this guide. At the end of this stage, you should have a fully functional CDC Java runtime environment. This stage is described in Chapter 3.

4. **Because object synchronization heavily affects runtime performance, it is necessary to tune the locking implementation.** Chapter 4 describes how to approach the task of finding a locking implementation that is appropriate for the target system. Several implementations are provided.

2.3   Source Code Organization

The reference source code for the CDC HotSpot Implementation Java virtual machine is organized into several top-level directories described in TABLE 2-1:

TABLE 2-1   Top-Level Directories

Directory	Description
build	CDC build system.
src	CDC source code.

2.3.1   `build` Directory

The CDC build system is based on a Linux or Solaris host development system using commonly available Java and UNIX development tools. The basic development model is to use the CDC build environment as a cross-compilation system to build an executable runtime environment. Then the executable runtime environment is loaded onto a target device for testing. For more information about how to install and use the CDC build environment, see the companion document *CDC Build System Guide*.

TABLE 2-2   `build` Directory

Directory	Description
<CPU>	CPU-specific makefiles
<OS>	OS-specific makefiles
<OS>-<CPU>	These makefiles contain low-level makefile macro definitions that are unique to the OS/CPU combination. For example, the location of a CPU-spcific version of `invokeNative.c` would be found here.
<OS>-<CPU>-<DEVICE>	The main build directory. This contains the top-level makefile with device-specific options and the generated target device-specific intermediate files for a Java runtime environment. These makefiles mostly set or override values used by the shared makefiles.
portlibs	Makefile definitions for the shared JIT layer.
share	Shared makefiles.

2.3.2 `src` Directory

The HPI source code is organized so that a small amount of target-specific implementation code supports the much larger shared source code. TABLE 2-3 describes the HPI source code hierarchy. These directories have parallel organizations to ease navigation and support the operation of the CDC build system.

The `src` directory contains the shared and target-specific source code for the CDC-HI Java virtual machine and both the CDC and Foundation class libraries. For porting, the most important directories are the HPI header files in `share/javavm/include/porting` and the HPI implementation files in `portlibs`, <**CPU**>`/javavm`, <**OS**>`/javavm` and <**OS**>-<**CPU**>`/javavm` directories. These directories are described in Chapter 3.

The organization described in TABLE 2-3 uses the following naming conventions: <**CPU**> is the CPU architecture, e.g. `arm`, `mips` or some other target CPU family. <**OS**> is the platform operating system, e.g. `linux`, `solaris` or some other platform operating system.

TABLE 2-3   `src` Directory

Directory	Language	Description
<CPU>		CPU architecture-specific source code. This code is shared by most ports based on <**CPU**>.
javavm	C	CPU-specific portion of the VM implementation. At the HPI level, this directory contains a small amount of assembly glue code as well as source for the dynamic compiler. This hierarchy performs a similar purpose to `src/portlibs`, but for a specific CPU-target. The files in this hierarchy use the *interface*_`cpu.[ch]` naming convention.
<OS>		OS-specific source code. This code is shared by all ports based on <**OS**>.
bin	C	OS-specific wrapper for the Java application launcher.
classes	Java	OS-specific portion of the CDC and Foundation Profile class libraries.

TABLE 2-3 `src` Directory

Directory	Language	Description
`javavm`	C	OS-specific portion of the VM implementation. These source files contain support functions that interact with the required OS services. The files in this hierarchy use the *interface*_`md.[ch]` naming convention.
`lib`	text	MIME content type system property table and platform-to-Java time zone mapping table.
`native`	C	JNI native methods for the CDC and FP class library source code. These native methods require porting for the target OS.
`tools`	C	OS-specific portion of the `hprof` profiler tool.
<OS>–<CPU>		OS/CPU-specific source code.
`javavm`	C	OS/CPU-specific portion of the VM implementation. These source files contain CPU-specific support functions that interact with low-level OS services like CPU-specific synchronization, numeric types and endianness. The files in this hierarchy use the *interface*_`arch.[ch]` naming convention.
`portlibs`	C	These porting utilities are used by most ports to map the HPI to common platform system interfaces. See TABLE 3-4 for a description of these porting utilities.
`share`[1]		Source code shred by different implementations.
`classes`	Java	CDC class library source code.
`foundation`	Java	Foundation Profile class library source code.
`javavm`	C	Shared portion of the VM implementation.

TABLE 2-3 `src` Directory

Directory	Language	Description
`javavm/include/porting`	C	These header files define the HPI interfaces that the target-specific source must implement. **NOTE:** The header files in this directory contain many descriptive comments that are not duplicated here. They should be studied carefully during the planning stages of a port.
`javavm/test`	Java	Various test programs for exercising the VM.
`lib`	text	Security policies.
`native`	C	JNI native methods for the CDC and FP class library source code. These native methods do not require porting.
`tools`	C	`hprof` source code.

1 The source code in the share hierarchy is intended for reference purposes only and should not be modified. If the default behavior needs modification, in most cases these changes should be made by overriding this default functionality with code in the target-specific source hierarchies. For other cases, see `https://javapart-ner.sun.com/partner/porting/04.html` for an overview of the review process for modifying shared code.

PART II HPI Layer

This part describes the HPI porting layer. This includes the CPU and OS-specific interfaces for both the virtual machine runtime and the CDC/Foundation Profile class library.

This part contains the chapters:

- Host Programming Interface
- Fast Locking

Host Programming Interface

This Host Programming Interface (HPI) represents the portability interface for the runtime component of the CDC HotSpot Implementation Java virtual machine. Implementing the macros, data structures and functions of the HPI is the first major stage in porting the CDC HotSpot Implementation Java virtual machine to a new target system.

This chapter cover the following topics:

- HPI Header File Hierarchy
- Creating an HPI Implementation

This chapter uses the Linux/ARM/Zaurus port to describe how an HPI implementation is structured. In general, we make the simplifying assumptions that the target port includes ANSI and POSIX libraries and that the CDC build system uses the patched `gcc` cross-compiler described in the companion document *CDC Build System Guide*. For help with porting projects that fall outside these guidelines, contact Java Partner Engineering (`http://www.sun.com/software/jpe`).

3.1 HPI Header File Hierarchy

The directory `src/share/javavm/include` contains a series of header files that impose a structure on an HPI port. The most important is `src/share/javavm/include/defs.h`, the top-level HPI include file that defines basic types and data structures as well as including the other HPI header files. The top-level HPI header file hierarchy is described in TABLE 3-1.

TABLE 3-1 HPI Header File Hierarchy

Header File	Description
`src/share/javavm/include/defs.h`	Defines basic types and data structures as well as including the other HPI header files.
`src/share/javavm/include/porting/defs.h`	Describes the file mapping for the ANSI and VM header files. Some advanced options are also described here. See TABLE 3-2 and TABLE 3-3 for descriptions of these header files.
`src/<OS>/javavm/include/defs_md.h`	The top-level target-specific HPI header file. Provides the file mapping for the ANSI and VM header files and selects advanced platform features as well as common macros used throughout the target-specific source code.
`src/<OS>-<CPU>/javavm/include/defs_arch.h`	OS-CPU architecture-specific macros.

3.1.1 CVM_HDR_* Header File Macros

The `CVM_HDR_*` header file macros described in `share/javavm/include/porting/defs.h` represent the basic requirements for an HPI port. These macros are usually defined in `<OS>/javavm/include/defs_md.h`.

There are two categories of header file macros:

- The `CVM_HDR_ANSI_*` macros described in TABLE 3-2 identify ANSI header files that define standard C library functions and data types used by the VM.

■ The rest of the CVM_HDR_* macros described in TABLE 3-3 identify the target-specific VM header files that define functions and data structures for accessing platform-level services. These are usually found in <**OS**>/javavm/include with corresponding implementation files found in <**OS**>/javavm/runtime. They are organized into the stages described in Section 3.2, "Creating an HPI Implementation" on page 3-5".

TABLE 3-2   ANSI Header File Macros

Macro	ANSI Header File	Description
CVM_HDR_ANSI_ASSERT_H	<assert.h>	Macro for verifying program assertion.
CVM_HDR_ANSI_CTYPE_H	<ctype.h>	Character types.
CVM_HDR_ANSI_ERRNO_H	<errno.h>	System error numbers.
CVM_HDR_ANSI_LIMITS_H	<limits.h>	Implementation-dependent constants.
CVM_HDR_ANSI_SETJMP_H	<setjmp.h>	Declarations for setjmp() and longjmp() that control transfers that bypass the normal function call and return protocol.
CVM_HDR_ANSI_STDARG_H	<stdarg.h>	Macros for handling variable argument lists.
CVM_HDR_ANSI_STDDEF_H	<stddef.h>	Standard type definitions.
CVM_HDR_ANSI_STDIO_H	<stdio.h>	Standard buffered I/O.
CVM_HDR_ANSI_STDLIB_H	<stdlib.h>	Standard library definitions.
CVM_HDR_ANSI_STRING_H	<string.h>	String operations.
CVM_HDR_ANSI_TIME_H	<time.h>	Time types.

The macros shown in TABLE 3-2 define the locations of the standard ANSI C header files. They can also be used to define an alternate location or implementation or to interpose a wrapper file around the system file. One reason to do this is to redefine tokens that are causing conflicts with the VM code or to include other system header files first.

In most cases, cloning and modifying the linux example should be sufficient since it's based on the porting libraries in portlibs/ansi_c and portlibs/gcc_32_bit.

TABLE 3-3   VM Header Files Macros

Macro	VM Header File	Description
CVM_HDR_DOUBLEWORD_H CVM_HDR_ENDIANNESS_H CVM_HDR_FLOAT_H CVM_HDR_INT_H	doubleword.h endianness.h float.h int.h	Data type macros and APIs.
CVM_HDR_GLOBALS_H	globals.h	VM global state.
CVM_HDR_MEMORY_H	memory.h	Memory access.
CVM_HDR_JNI_H CVM_HDR_LINKER_H	jni.h linker.h	Native method support and dynamic linking support.
CVM_HDR_THREADS_H	threads.h	Thread support.
CVM_HDR_SYNC_H	sync.h	Synchronization support.
CVM_HDR_IO_H CVM_HDR_PATH_H CVM_HDR_SYSTEM_H CVM_HDR_TIME_H CVM_HDR_TIMEZONE_H	io.h path.h system.h time.h timezone.h	I/O support, Java system properties, time and system functions for halt and reset.
CVM_HDR_NET_H	net.h	Networking support.

3.1.2   src/portlibs Porting Libraries

The src/portlibs directory contains many useful porting libraries that simplify the task of porting the HPI layer to most platforms.

TABLE 3-4   src/portlibs Sub-Directories

Directory	Description
ansi_c	Maps internal data types used by the HPI to common ANSI C data types.
dlfcn	Supports dynamic linking of shared libraries directly through the dlsym() system call.
gcc_32_bit	Maps internal data types used by the HPI to common gcc data types.
jit	RISC porting library for the dynamic compiler.
posix	I/O, socket and thread porting functions based on POSIX.
realpath	UNIX path name handling.

3.2 Creating an HPI Implementation

In this section we outline the steps necessary for creating an HPI implementation. Most of these steps are straightforward. A few require additional effort for which there is help in the form of porting libraries and examples.

Developing a new port from scratch is probably more work than is necessary. The sample Linux/ARM/Zaurus port serves two purposes. It's well structured and commented so that it can be used as a good example for porting. And it's also well tested and optimized for production use. Java Partner Engineering can provide other example ports, and the porting libraries in `src/portlibs` should simplify porting for most common target platforms.

Java Partner Engineering (`http://www.sun.com/software/jpe`) has ports available for other operating systems and CPUs. Since the HPI porting layer is separated into CPU and OS abstractions, the example for one OS support layer can be used as a starting point for a port to another target system. That is, the same OS layer can be used even if it's based on a different CPU. Starting with one of these prebuilt modules will save a great deal of effort in porting the CDC HotSpot Implementation Java virtual machine to a new target platform.

TABLE 3-5 shows the CPU ports available from Java Partner Engineering.

TABLE 3-5 CPU Ports

CPU	Support Level
ARM	V4 architecture and higher
MIPS	MIPS II, III, IV, and V
PowerPC	PowerPC and e500 architectures
Sparc	MicroSparc and UltraSparc
x86 (interpreter only)	386 and higher

TABLE 3-6 shows the operating system ports available from Java Partner Engineering.

TABLE 3-6   Operating System Ports

Operating Systems	CPU	Devices
Linux	ARM	• Sharp Zaurus • Rebel.Com Netwinder • iPaq running Familiar Linux • MontaVista Linux • Intel Bulverde chipset on the Mainstone development board • TI Innovator development board for the OMAP platform
	MIPS	• Cobalt Raq2 and Qube2 running Debian Linux 3.0.1 • SGI Indy and Indigo2 running Debian Linux 3.0.1 • MontaVista Linux
	PowerPC	• YellowDog Linux 2.2, 2.3, and 3.0 on an Apple PowerMac • MontaVista Linux
	SPARC	• Sun UltraSparc running Debian Linux 3.0.1
	x86	• x86/PC running Red Hat Linux 7.2 and 9.0
Darwin (BSD-based)	PowerPC	PowerMac running MacOS 10.*x*
Solaris	SPARC	Solaris 2.7, 2.8, and 2.9 on Sun UltraSPARC hardware
Win32	ARM	WinCE 3.0 on PocketPC (iPaq)
	MIPS	WinCE 4.1 on development board
	x86	Windows 2000

3.2.1   Suggested Work Flow

The work flow for developing an HPI implementation is divided into the stages listed below.

1. **Section 3.2.2, "Prepare the Target-Specific `build` and `src` Hierarchies" on page 3-7** describes how to setup the build and src hierarchies by cloning or combining code from existing ports.

2. **Section 3.2.3, "Data Types, Global State and Memory Access Support" on page 3-8** represents the main portion of the port. The goal is to implement enough of the HPI to perform basis tests that don't require I/O capability.

3. **Section 3.2.4, "JNI Support" on page 3-9** adds dynamic linking and JNI support. The goal is to enable native method support required by the Java class library.

4. **Section 3.2.5, "Thread Support" on page 3-10** adds thread support for both VM operation and Java runtime support.

5. **Section 3.2.6, "Synchronization Support" on page 3-11** add synchronization support.

6. **Section 3.2.7, "I/O and System Support" on page 3-12** adds I/O capability to the basic port to enable local file system-based class loading and I/O.

7. **Section 3.2.8, "Networking Support" on page 3-12** adds support for socket-based networking to enable network-based class loading and I/O.

Each section below includes a description of the source code for the Linux/ARM/Zaurus sample implementation.

3.2.2 Prepare the Target-Specific `build` and `src` Hierarchies

The steps below are based on a target platform based on the operating system `MyOS`, the CPU `MyCPU` and the device `MyDevice`. The example is based on cloning the Linux/ARM/Zaurus sample implementation. See TABLE 3-5 and TABLE 3-6 for descriptions of other ports available from Java Partner Engineering that can be used to more closely match the target device.

1. **Prepare the UNIX build tools for the new target device.**

This will require configuring and building the cross-development build tools to include code generation capabilities for the new target device. See Chapter 2 and the companion document *CDC Build System Guide* for more information about configuring the CDC build system.

2. **Refactor the platform-dependent makefiles hierarchy to support the target platform.**

If necessary, this involves cloning four directories. For example,

```
% cp -r build/linux-arm-zaurus build/MyOS-MyCPU-MyDevice
% cp -r build/linux-arm build/MyOS-MyCPU
% cp -r build/linux build/MyOS
% cp -r build/arm build/MyCPU
```

Note that these cloning steps are only necessary for build directories that will be modified. If the target CPU or OS is one of the ports described in TABLE 3-5 or TABLE 3-6, then no modification should be necessary at this stage of the port.

> **Tip –** If the target OS and/or CPU are among the ports listed in TABLE 3-5 and TABLE 3-6, then use or modify the prebuilt makefiles instead.

3. **Edit** `build/MyOS-MyCPU-MyDevice/GNUmakefile` **to define any necessary compiler flags and build-time options.**

4. **Refactor the platform-dependent source code hierarchy.**

```
% cp -r linux-arm MyOS-MyCPU
% cp -r linux MyOS
% cp -r arm MyCPU
```

The `MyOS` directory will contain the `bin`, `lib` and `tools` sub-directories for the Java application launcher, system property files and profiler tools respectively. These should not require modification for a Linux port. Again, source code from other ports may be more appropriate than the Linux/ARM/Zaurus example used here.

Note that these cloning steps are only necessary for source directories that will be modified. If the target CPU or OS is one of the ports described in TABLE 3-5 or TABLE 3-6, then no modification should be necessary at this stage of the port.

5. **For each of the stages outlined in the following sections, supply the required implementations.**

For most target platforms, there should be some code supplied by one of the ports described in TABLE 3-5 or TABLE 3-6. So the actual amount of implementation steps should be a subset of the steps described below.

> **Tip –** For ports that use existing OS and CPU implementations, for example porting `linux-arm-zaurus`, the only work requried is cleaning up the target device build directory and editing the `GNUmakefile`.

3.2.3 Data Types, Global State and Memory Access Support

The first stage of implementing the HPI layer is to implement the data types and support functions.

Create target-specific implementation source files for the interfaces shown in
TABLE 3-7. Again, the header files in share/javavm/include/porting include
comments that describe the required interfaces. The existing ports use the porting
libraries in src/portlibs.

TABLE 3-7 Implementation Source Files for Data Types, Global State and Memory Access
Support

Header File	Examples
porting/doubleword.h	linux/javavm/include/doubleword_md.h linux-arm/javavm/include/doubleword_arch.h portlibs/gcc_32_bit/doubleword.h portlibs/ansi_c/doubleword.h
porting/endianness.h	linux/javavm/include/endianness_md.h linux-arm/javavm/include/endianness_arch.hlinux-arm/javavm/include/endianness_arch.h
porting/float.h	linux/javavm/include/float_md.h linux-x86/javavm/include/float_arch.hlinux-arm/javavm/include/float_arch.h arm/javavm/runtime/arm_float_cpu.c portlibs/gcc_32_bit/float.h portlibs/ansi_c/float.h
porting/int.h	linux/javavm/include/int_md.h linux-arm/javavm/include/int_arch.h portlibs/ansi_c/int.h
porting/globals.h	linux/javavm/include/globals_md.h linux/javavm/runtime/globals_md.c
porting/memory.h	linux/javavm/include/memory_md.h linux-arm/javavm/include/memory_arch.h arm/javavm/runtime/atomic_arm.S arm/javavm/runtime/memory_asm_cpu.S[1]

1 memory_asm_cpu.S contains ARM-specific optimizations that are not necessary for general-purpose porting.
These APIs can be deactivated by removing the related macros in memory_arch.h.

3.2.4   JNI Support

The next stage is to add dynamic linking and JNI support. The sample
Linux/ARM/Zaurus implementation uses Linux shared libraries to provide native
library implementations. Many UNIX-like operating systems provide similar
features.

Create target-specific implementation source files for the interfaces shown in
TABLE 3-8.

TABLE 3-8 Implementation Source Files for JNI Support

Header File	Linux/ARM/Zaurus **Example**

`porting/jni.h`	`portlibs/gcc_32_bit/jni.h`
	`linux/javavm/include/jni_md.h`
	`arm/javavm/runtime/`
	` invokeNative_arm.S`

`porting/linker.h`	`portlibs/dlfcn/linker_md.c`
`share/native/common/jni_statics.h`	`linux/native/common/statics_md.h`

3.2.4.1   CVMjniInvokeNative

The most important part of this stage is the implementation of the
`CVMjniInvokeNative()` routine which translates Java method calling conventions
into the platform language (usually C) calling conventions used by native methods.
The Java VM passes all the arguments in the Java stack and expects the results to be
placed there as well.

For performance and stack safety reasons the implementation of
`CVMjniInvokeNative()` should be written in assembly language. The comments
in `src/share/javavn/include/porting/jni.h` and
`arm/javavm/runtime/invokeNative_arm.S` describe the interface and
implementation of `CVMjniInvokeNative()`. Contact Java Partner Engineering
(`http://www.sun.com/software/jpe`) to get versions of
`CVMjniInvokeNative()` for alternate processors.

3.2.5   Thread Support

If the target system has a POSIX thread library, then porting the thread support
portion of the HPI is straightforward. In fact,
`src/portlibs/posix/posix_threads_md.c` contains such an implementation.
The alternatives are to find and port a POSIX thread library to the native platform or
to port the thread interface directly to the native interface.

The `porting/defs.h` header file describes the `CVM_HAVE_PROCESS_MODEL` option
that indicates that the target platform provides a process model. This allows the VM
to avoid waiting for daemon threads to exit before shutting down.

Multi-processor systems are not supported by the CDC HotSpot Implementation
Java virtual machine unless all CDC threads can be isolated to the same processor.

Create target-specific implementation source files for the interfaces shown in TABLE 3-9.

TABLE 3-9   Implementation Source Files for Thread Support

Header File	Linux/ARM/Zaurus Example

`porting/threads.h`	`portlibs/posix/threads.h`
	`portlibs/posix/posix_threads_md.c`
	`linux/javavm/include/threads_md.h`
	`linux/javavm/runtime/threads_md.c`
	`linux-arm/javavm/include/threads_arch.h`

3.2.6   Synchronization Support

The speed of object synchronization greatly affects overall runtime performance. The default locking mechanism will work without porting effort, but is slow. Choosing between the other alternatives will require experimentation with the target platform. This is by far the most complex part of an HPI port to understand, though implementation is made easier by the options described in Chapter 4.

The `porting/sync.h` header file describes other advanced options:

- `CVM_ADV_SCHEDLOCK` requires implementations of the `CVMschedLock()` and `CVMschedUnlock()` functions.
- `CVM_ADV_MUTEX_SET_OWNER` requires the implementation of `CVMmutexSetOwner()`.
- `CVM_ADV_THREAD_BOOST` enables thread priority boosting. This option requires implementations of the `CVMthreadBoostInit()`, `CVMthreadAddBooster()`, `CVMthreadBoostAndWait()` and `CVMthreadCancelBoost()` functions.

Create target-specific implementation source files for the interfaces shown in TABLE 3-10.

TABLE 3-10   Implementation Source Files for Synchronization Support

Header File	Linux/ARM/Zaurus Example

`porting/sync.h`	`linux/javavm/include/sync_md.h`
	`linux/javavm/runtime/sync_md.clinux-`
	`arm/javavm/runtime/sync_arch.c`
	`linux-arm/javavm/include/sync_arch.h`
	`linux-arm/javavm/runtime/sync_arch.c`
	`arm/javavm/runtime/atomic_arm.S`
	
	`portlibs/posix/sync.h`
	`portlibs/posix/posix_sync_md.c`

3.2.7   I/O and System Support

The next stage is to add I/O and system support. If the target system has a POSIX-like I/O toolkit, then this stage is straightforward.

Create target-specific implementation source files for the interfaces shown in TABLE 3-11.

TABLE 3-11   Implementation Source Files for I/O Support

Header File	Linux/ARM/Zaurus **Example**

`porting/io.h`	`linux/javavm/include/io_md.h`
	`linux/javavm/runtime/io_md.c`
	`portlibs/posix/io.h`
	`portlibs/posix/posix_io_md.c`

`porting/path.h`	`linux/javavm/include/path_md.h` `portlibs/realpath/canonicalize_md.c`
`porting/java_props.h`	`linux/javavm/runtime/java_props_md.c` `linux/javavm/runtime/locale_str.h`
`porting/system.h`	`linux/javavm/runtime/system_md.c`

`porting/time.h`	`linux/javavm/include/time_md.h`
	`linux/javavm/runtime/time_md.c`
	`portlibs/posix/posix_time_md.c`

`porting/timezone.h`	`linux/javavm/runtime/timezone_md.c`

3.2.8   Networking Support

The last stage is to implement socket-based network support. Again, using a POSIX library will simplify this task.

Create target-specific implementation source files for the interfaces shown in TABLE 3-12.

TABLE 3-12   Implementation Source Files for Networking Support

Header File	Linux/ARM/Zaurus **Example**
`porting/net.h`	`linux/javavm/include/net_md.h` `linux/javavm/runtime/net_md.c`

3.3　CDC Class Library Support Layer

The CDC class library support layer represents the portability interface for the CDC Java class library. This portability layer is divided into a small amount of C source code and a few low-level Java classes that interact with system-level services like a socket-based network stack, a file system and a native process model.

Implementing the macros, data structures and functions of this layer is based on providing native method implementations for certain system level classes in the CDC Java class library. The platform-level portion of the JNI mechanism is provided the HPI functions described in Section 3.2.4, "JNI Support" on page 3-9. For most UNIX-like platforms, the Linux-based platform-level Java classes should require little, if any, modification.

This chapter shows how to implement the system-level native methods for a new target system. It is divided into the following sections.

- Section 3.3.1, "Source Code Organization" on page 3-13" describes the HPI header files and the sample implementations.
- Section 3.3.2, "Creating a CDC Class Library Support Layer Implementation" on page 3-15" describes the basic workflow for porting the class library support layer.

3.3.1　Source Code Organization

The source code organization for the class library support layer is based on a combination of the Java class library and the platform-specific source code for the CDC reference implementations. For example, the CDC reference implementations keep operating system specific source code in `src/`*OS*. And the source code for the `java.net` package is usually kept in a hierarchy with a `java/net` directory structure. So for the Linux/ARM/Zaurus implementation, the native method implementations for the `java.net` package are kept in `src/linux/native/java/net`.

Profile-based Java and JNI source code can be found in `src/*/profile`.

TABLE 3-13 describes the native method implementations for the CDC class library support layer in `src/linux/native/java`. There are three packages that require native method support: `java.io`, `java.lang` and `java.net`.

TABLE 3-13 JNI Native Method Source Code in `src/linux/native/java`

Source Files	Description
`io/FileSystem_md.c` `io/UnixFileSystem_md.c`	The platform file system classes in `java.io` are based on common POSIX-based file system semantics. The native methods in `src/linux/native/java/io` are POSIX-based. Other file system types are possible. See `src/share/classes/java/io/FileSystem.java`.
`lang/Runtime_md.c` `lang/UNIXProcess_md.c`	`java.lang.Process` provide access to platform-level processes. These should be very similar for most UNIX-like platforms.

`net/InetAddressImpl_md.c`
`net/PlainDatagramSocketImpl_md.c`
`net/PlainSocketImpl_md.c`
`net/SocketInputStream_md.c`
`net/SocketOutputStream_md.c`
`net/net_util_md.c`
`net/net_util_md.h`
	`java.net` is based on Berkeley sockets available on most UNIX-like platforms. The native methods in `src/linux/native/java/net` are based on the socket interface. See TABLE P-2 for a reference document for the Berkeley Socket interface.

The `src/share/native` directory contains several portable native method implementations that do not require modification.

TABLE 3-14 describes platform-level Java classes in the `linux/classes` directory.

TABLE 3-14 Platform-Level Java Classes in `linux/classes`

Source Files	Description
`java/io/UnixFileSystem.java`	UNIX file system support.
`java/lang/Terminator.java` `java/lang/UNIXProcess.java`	UNIX process support.
`sun/net/www/protocol/file/` `  Handler.java`	`file` protocol handler.
`sun/net/www/protocol/jar/` `  JarFileFactory.java`	Retrieving and caching `jar` archives.

3.3.2 Creating a CDC Class Library Support Layer Implementation

Porting the CDC class library support layer is very straightforward if the target system provides the baseline platform requirements:

- POSIX I/O toolkit
- ANSI standard C library
- Berkeley sockets

As with the HPI layer, there are several existing ports that can be used as a starting place for porting. For example, the Darwin and Solaris ports provide good examples for most BSD-flavored UNIX implementations. See TABLE 3-5 and TABLE 3-6 for a list of CPU and operating system ports available from Java Partner Engineering.

Porting the CDC class library support layer to platforms that don't provide these resources requires much more effort. The best way to approach that task is to understand these interfaces and to either find and port compatibility libraries or write native method implementations that map platform-level services into the more standard functionality required by the CDC class library support layer. Java Partner Engineering can help with these non-standard ports by providing example code and consulting services.

3.4 Simple Test Procedure

At this point the CDC Java runtime environment should be testable with following basic procedure. The companion documents *CDC Runtime Guide* and *CDC Build System Guide* have more information about how to exercise the CDC Java runtime environment.

1. **Build the CDC Java runtime environment.**

   ```
   % make CVM_JIT=false
   ```

2. **Test the CDC Java runtime environment.**

   ```
   % bin/cvm -cp testclasses.zip HelloWorld
   ```

   This will provide a basic test for I/O support.

CHAPTER **4**

Fast Locking

Fast locking is a speed optimization technique for reducing the time needed to do object synchronization. Because the vast majority of locking is uncontended, this optimization is achieved by introducing the use of a lightweight lock in the absence of contention for a lock between threads. A lightweight lock is a data structure that records the fact that an object was locked instead of actually binding a monitor to the object and locking the monitor.

If no contention occurs, locking and unlocking of an object is achieved by simply marking the object's lightweight lock data structure accordingly. If contention occurs, the object will then be inflated by binding a monitor to the object. This is done by the contending thread, and ownership is assigned to the owner thread. After inflation, locking and unlocking of the object is done by actually locking and unlocking the monitor which is now bound to the object. A monitor object tends to be heavyweight because it normally involves system calls.

At some point in time, for example during garbage collection, an unlocked object monitor may be deflated. That means if the object is not locked at that time, its monitor may unbind from that object. After deflation, locking and unlocking of the object will be go through the lightweight lock mechanism again until contention occurs. Because locking an object through the lightweight lock mechanism generally takes less time and resources than the heavyweight, inflated mechanism, this scheme results in faster object synchronization.

4.1   Fast Lock Implementations

Each object has a header which contains a word of bits indicating the current state of the object from a locking perspective. The possible states are:

- *Locked.* The locked state indicates that the object is locked with a lightweight lock and is associated with a lock record data structure which contains further information about the state of the object.

- *Inflated into a heavyweight monitor.* The inflated state indicates that the object has been inflated and is bound to a monitor as well as a lock record data structure. Both the monitor and lock record contain information about the object's state.
- *Unlocked.* The unlocked state indicates that the object is unlocked and is not associated with any lightweight lock nor heavyweight monitor. The state of the object is contained entirely in the object header itself.

The object header word bits, the lock record and any bound monitor must be manipulated under atomic conditions.

The CDC HotSpot Implementation Java virtual machine provides several options to implement this atomicity, as expressed in the possible values for the CVM_FASTLOCK_TYPE option described in . These options are usually set in the platform sync_arch.h header file because the OS and CPU both play a role in determining the proper CVM_FASTLOCK_TYPE. For example, in the Linux/ARM example, this is in linux-arm/javavm/include/sync_arch.h.

TABLE 4-1   CVM_FASTLOCK_TYPE Values

Option	Description
CVM_FASTLOCK_NONE	The fast locking technique of object synchronization will not be used. Every time an object is to be locked, the object will be inflated and bound to a monitor. Locking and unlocking will always be done through the monitor. If CVM_FASTLOCK_TYPE is set to CVM_FASTLOCK_NONE, the platform does not need to define CVM_ADV_MUTEX_SET_OWNER. This is the easiest fast locking technique to port, but it is also the slowest.

CVM_FASTLOCK_ATOMICOPS	The atomicity is achieved using atomic compare-and-swap and atomic swap operations. The platform must define the CVM_ADV_ATOMIC_CMPANDSWAP and CVM_ADV_ATOMIC_SWAP options and provide implementations for:
	
	`CVMUint32 CVMatomicCompareAndSwap(volatile`
	`  CVMUint32 *addr , CVMUint32 new, CVMUint32 old);`
	`CVMUint32 CVMatomicSwap(volatile CVMUint32 *addr,`
	`  CVMUint32 new);`
	
	The platform must also define the CVM_ADV_MUTEX_SET_OWNER option and provide an implementation for:
	
	`void CVMmutexSetOwner(CVMThreadID *self,`
	`  CVMMutex* m , CVMThreadID *ti);`

CVM_FASTLOCK_MICROLOCK	The atomicity is achieved using microlocks which are essentially non-reentrant mutexes that are used to protect a critical region of code. The values of the CVM_MICROLOCK_TYPE option indicate the available microlock implementations:
	
	• CVM_MICROLOCK_DEFAULT - The microlocks will be implemented using the platform's implementation of CVMMutex.
	
	Alternatively, the platform could choose to define the CVM_HAVE_PLATFORM_SPECIFIC_MICROLOCK option. If so, the platform will have to provide implementations for all the CVMMicroLock APIs.
	
	• CVM_MICROLOCK_SCHEDLOCK - The microlocks will be implemented using a lockout of the platform's thread scheduler. As such, the platform must define the CVM_ADV_SCHEDLOCK option and provide implementations for:
	`void CVMschedLock(void);`
	`void CVMschedUnlock(void);`
	
	The platform must also define CVM_ADV_MUTEX_SET_OWNER option and provide an implementation for:
	`void CVMmutexSetOwner(CVMThreadID *self,`
	`  CVMMutex* m , CVMThreadID *ti);`

4.2 Choosing a Fast Lock Implementation

Deciding which fast lock implementation to use depends on the availability of certain platform-level APIs and the target processor's atomic instructions.

It's easiest to start by setting `CVM_FASTLOCK_TYPE` to `CVM_FASTLOCK_NONE`. This option allows the CDC HotSpot Implementation Java virtual machine to be ported with the minimal amount of effort spent on object synchronization issues. The platform must still provide implementations for the `CVMMutex` APIs. However, the implementation of `CVMmutexSetOwner` will not be necessary. Keep in mind that this implementation is probably the slowest and most resource intensive of all the fast locking implementations.

If the platform can provide an implementation of `CVMmutexSetOwner`, the next easiest combination is:

```
#define CVM_FASTLOCK_TYPE CVM_FASTLOCK_MICROLOCK
#define CVM_MICROLOCK_TYPE CVM_MICROLOCK_DEFAULT
#undef CVM_HAVE_PLATFORM_SPECIFIC_MICROLOCK
```

This combination of options allows the CDC HotSpot Implementation Java virtual machine to use the fast locking technique without the additional burden of having the platform provide the implementation of `CVMmutexSetOwner`.

Next, consider which implementation provides the fastest implementation. This can only be determined by trying each implementation on the platform. The reason there is no steadfast rule as to which implementation is faster is because the speed performance of the implementation depends on the platform's implementation of the underlying supporting APIs as indicated in .

Generally, setting `CVM_FASTLOCK_TYPE` to `CVM_FASTLOCK_ATOMICOPS` provides the fastest implementation. However, this is dependent on the platform being able to provide implementations for `CVMatomicCompareAndSwap` and `CVMatomicSwap`.

If these atomic operations are not available on the platform, the next fastest implementation may be `CVM_FASTLOCK_MICROLOCK`. However, the default implementation of microlocks in the CDC HotSpot Implementation Java virtual machine uses `CVMMutex`. Doing this may make `CVM_FASTLOCK_MICROLOCK` slower than `CVM_FASTLOCK_NONE` on some platforms.

If the platform can provide some fast alternate mechanism of achieving mutual exclusion, then it is possible to redefine the implementation of the VM's microlocks to use this fast mechanism. The `CVM_MICROLOCK_SCHEDLOCK` option is provided as a way to implement microlocks if the platform has the mechanism for disabling the

platform scheduler from doing any context switches. Alternatively, the platform can define `CVM_HAVE_PLATFORM_SPECIFIC_MICROLOCK` and provide its own implementation of the `CVMMicroLock` APIs.

4.3 Implementations

TABLE 4-2 summarizes the `CVM_FASTLOCK_TYPE` and `CVM_MICROLOCK_TYPE` choices for the RIOI ports currently available through Java Partner Engineering..

TABLE 4-2   Fast Lock Implementations

Port	CVM_FASTLOCK_TYPE	CVM_MICROLOCK_TYPE
Linux/ARM	MICROLOCK	DEFAULT(spinlock)
Linux/MIPS	ATOMICOPS	DEFAULT(mutex)
Linux/PowerPC	ATOMICOPS	DEFAULT(mutex)
Linux/SparcV8	MICROLOCK	DEFAULT(spinlock)
Linux/SparcV9	ATOMICOPS	DEFAULT(mutex)
Linux/x86	ATOMICOPS	DEFAULT(mutex)
WinCE/ARM	ATOMICOPS	DEFAULT(mutex)
WinCE/MIPS	ATOMICOPS	DEFAULT(mutex)
Win32/X86	ATOMICOPS	DEFAULT(mutex)
Solaris/SparcV8	MICROLOCK	DEFAULT(spinlock)
Solaris/SparcV9	ATOMICOPS	DEFAULT(mutex)
Darwin/PowerPC	ATOMICOPS	DEFAULT(mutex)

The following notes describe how these choices were made. This should provide help with choosing appropriate values for `CVM_FASTLOCK_TYPE` and `CVM_MICROLOCK_TYPE` for a target platform.

Normally if `CVM_MICROLOCK_DEFAULT` is specified, a mutex-based microlock implementation is provided. However, this can be very slow if microlocks are being used as the fastlock type (`CVM_FASTLOCK_MICROLOCK`). In this case the platform normally will `#define CVM_HAVE_PLATFORM_SPECIFIC_MICROLOCK` and provide its own microlock implementation in the form of a spinlock. However, spinlock-based microlocks are only safe if threads don't strictly enforce priorities. This is why they cannot be used for WinCE.

All the platforms above except for WinCE could have chosen to use a spinlock-based microlocks rather than the default mutex-based microlocks. However, the benefit of doing so is insignificant if CVM_FASTLOCK_MICROLOCK is not being used, since microlocks are rarely used in this case.

Note that even if CVM_FASTLOCK_TYPE is not CVM_FASTLOCK_MICROLOCK, microlocks will still used by the shared code for purposes other than object synchronization. Since these represent a small portion of actual usage, it does not affect overall performance.

Setting CVM_FASTLOCK_TYPE can be summarized as follows: If a CAS instruction is available, use CVM_FASTLOCK_ATOMICOPS. Otherwise use CVM_FASTLOCK_MICROLOCK. Note that SPARC v8 (also known as microSparc) does not have a CAS instruction but SPARC v9 does. This is why the SPARC v8 uses CVM_FASTLOCK_MICROLOCK and SPARC v9 uses CVM_FASTLOCK_ATOMICOPS.

Setting CVM_MICROLOCK_TYPE can be summarized as follows: If a reasonably fast thread schedlock is available, use CVM_MICROLOCK_SCHEDLOCK. Otherwise use CVM_MICROLOCK_DEFAULT. If CVM_MICROLOCK_DEFAULT is used, then if CVM_FASTLOCK_MICROLOCK is also used, #define CVM_HAVE_PLATFORM_SPECIFIC_MICROLOCK and provide a spinlock implementation, assuming the OS thread scheduling allows for this (no strict thread priorities).

WinCE/ARM is an exception for the normal approach used to determine the proper locking types. At first is appears that it should use CVM_FASTLOCK_MICROLOCK since ARM does not have a CAS instruction. However, since a spinlock microlock implementation cannot be used on WinCE (because of strict thread priorities), the fastlock implementation would be done via a muxtex-based microlock, which would be slow. WinCE does provide a CAS OS call. Although rather slow compared to an actual CAS instruction, it is better than the alternative of using CVM_FASTLOCK_MICROLOCK with a mutex based microlock implementation.

PART III Dynamic Compiler Layer

This part describes the porting interfaces for the dynamic compiler.

Dynamic Compiler

Dynamic Compiler

The CDC HotSpot Implementation Java virtual machine includes a dynamic compiler that can be easily ported to different target RISC CPUs. This chapter describes the dynamic compiler's portability interface and shows how a port is structured.

This chapter cover the following topics:

- Dynamic Compiler Overview
- Dynamic Compiler Header File Hierarchy

It may be helpful to become familiar with the compiler policy command-line options described in the companion document *CDC Runtime Guide*.

5.1   Dynamic Compiler Overview

This section provides a short description of the dynamic compiler's structure and operation. The purpose of this overview is to introduce terms and provide context for the porting effort, not to provide a theory of operations for the dynamic compiler itself. It is important to note that only a small well-defined portion of the dynamic compiler requires attention during the porting process. Most of the implementation is in shared source code that does not require review or modification during the porting process. On the other hand, a sophisticated knowledge of the target CPU architecture is necessary.

TABLE 5-1 describes the modules of the dynamic compiler.

TABLE 5-1   Dynamic Compiler Modules

Module	Implementation	Description
Compilation policy	shared	A mechanism for choosing what and how to compile. The dynamic compiler provides command-line options for expressing a compilation policy at runtime. See the companion document *CDC Runtime Guide* for a description of these command-line options.
Front-end (IR generator)	shared	Once a method has been selected for compiling, the front-end translates its bytecodes into an intermediate representation (IR) which represents expressions with a directed acyclical graph (DAG) data structure that simplifies optimization and code generation. Semantically, the IR is at a slightly lower level than bytecodes and uses an idealized RISC instruction set that allows for some parameterization. So during the first pass many code optimizations are performed by shared code, the most important being inlining.
JavaCodeSelect	shared	JavaCodeSelect is a parser generator similar to YACC. Where YACC is a general-purpose parser generator based on pattern matching within streams, JavaCodeSelect produces a parse that performs pattern matching within tree-based data structures. JavaCodeSelect runs at build-time and generates a pattern matching parser for the back-end code generator.
Back-end code generator	some porting	The back-end code generator translates the method's optimized IR representation into a native instructions for storage in the code buffer and eventual execution on the target CPU. Most of the porting effort for the back-end is in the implementation of emitter functions that generate native bit encodings for each IR node. The back-end also contains a code generation engine built with JavaCodeSelect. After the initial IR translation, this code generation engine translates the IR representation into calls to the emitters. Most of the JavaCodeSelect grammar rules are written in shared code and do not require porting. A more advanced port could override some of these rules.[1]
Compiled code manager (CCM)	some porting	The CCM provides pre-optimized static code routines for representing complex bytecode fragments that are difficult to compile dynamically. These are sometimes called *helper functions*. All of the helper functions have shared C implementations that can be replaced with optimized assembly code.

TABLE 5-1   Dynamic Compiler Modules

Module	Implementation	Description
Register manager	some porting	The register manager controls resource objects that can be used to store expression results and other semantic actions by the optimizer or back-end. These resource objects may be either a physical register or a spill area in the Java stack frame. The porting effort for the register manager is based on the CPU abstraction interface that identifies which register sets that the register manager can and cannot use.
Stack manager	shared	The stack manager is in charge of compile-time management of the Java expression stack and parameter stack. It keeps track of the expressions currently on the stack, which among other things, are used to compute stackmaps for GC safe-points.
Glue code	some porting	Target specific code (in assembler) that expedites calling from dynamically compiled code to CDC-HI runtime support routines such as CCM helper functions.
Code cache management	shared	Once a method has been compiled, it is stored in the code cache so that the interpreter or other compiled methods can access it.

1 Overriding the default `JavaCodeSelect` grammar rules is not currently supported in this porting guide.

As shown in TABLE 5-1, most of the porting effort is condensed into some initialization functions, and groups of well-defined emitter and helper functions.

There are basically three layers to the dynamic compiler implementation:

- Most of the shared code in `src/share` is target-independent.
- The code in `src/portlibs/risc` is a RISC porting library that abstracts the common features of RISC architectures.
- The code in `src/<`**CPU**`>` and `src/<`**OS**`>-<`**CPU**`>` contain target-specific code.

5.2   Dynamic Compiler Header File Hierarchy

The directory `share/javavm/include/porting/jit` contains the top-level header files that describes the macros, data structures and functions required by a port. These header files are described in TABLE 5-2.

TABLE 5-2   Dynamic Compiler Header File Hierarchy

Header File	Description
`share/javavm/include/porting/jit/` `   jit.h`	Top-level porting layer for the dynamic compiler. This header file describes macros that specify target-specific capabilities, data structures and support functions provided by the target implementation. Implementations for most of these support functions are available in the RISC porting library in `src/portlibs/jit/risc`.
`portlibs/jit/risc/include/porting/` `   jitrisc.h`	The CPU abstraction interface defines the characteristics of the CPU.
`portlibs/jit/risc/include/porting/` `   jitriscemitter.h`	Emitter porting layer.

`share/javavm/include/porting/jit/`
`   ccm.h`
`portlibs/jit/risc/include/porting/`
`   ccmrisc.h`
	Helper porting layer.

5.2.1   `portlibs/jit/risc` RISC Porting Library

The `portlibs/jit/risc` directory contains a RISC porting library that is the basis of CDC-HI's shared RISC porting strategy.

TABLE 5-3   `portlibs/jit/risc` RISC Porting Library

File/Directory	Description
`ccmintrinsics_risc.c`	Intrinsics for a few methods in `java.lang`.
`jit_risc.c`	Back-end.
`jitemitter.c`	This file implements all of the conditional emitters, and is meant for platforms that don't support conditional instructions other than branches.
`jitopcodes.c` `jitopcodes.h`	Shorthand names for the opcode values used by the back-end.
`jitregman.c`	Register manager.
`jitstackman.c` `jitstackman.h`	Stack manager.
`jitgrammar.h`	Data structures for the codegen-time expression/semantic stack.

`jitfloatgrammarrules.jcs`
`jitgrammardefs.jcs`
`jitgrammarincludes.jcs`
`jitgrammarrules.jcs`
	`JavaCodeSelect` input files that contain the default shared grammar rules.

`include/export`	Public interface to be used by CPU-specific code emitters.
`include/porting`	Portability interfaces to be implemented by CPU-specific ports.

5.3   Creating a Dynamic Compiler Implementation

In this section we outline the steps necessary for creating a dynamic compiler implementation. As with the HPI, other ports are available from Java Partner Engineering that can be used as examples. The best approach is to choose a CPU from the list in with similar features to the target CPU. Then use the

techniques found in that port as an example for the target port. This will help with the initial stages of porting, but nothing can replace the research/testing/tuning cycle necessary to exploit the best features of the target CPU.

5.3.1   Suggested Work Flow

1. **Get the initial port working.**

   a. **Collect architectural characteristics.** This includes defining macros for the CPU abstraction interface that identify available registers, maximum load/store offsets and whether the target CPU has conditional ALU instructions.

   b. **Implement some glue code.** Many glue functions are short pieces of assembly code that bind compiled code to C helper functions.

   c. **Implement the invoker functions.** The four invoker functions described in `share/javavm/include/porting/jit/ccm.h` are critical to system performance. These are usually written in assembly language.

   d. **Implement the emitter functions.** Because the emitter and helper functions are well defined they can be isolated for straightforward optimization and testing.

   These steps should require about 4500 lines of code: a small amount for the CPU abstraction interface, about 1500 lines for the glue code and about 3000 lines for the emitter functions.

2. **Tune the implementation by supplying optimized helper function implementations.** Software floating point support is also implemented through a separate group of helper functions.

5.3.2   CPU Abstraction Interface

The first stage is to write some macros described in
`portlibs/jit/risc/include/porting/jitrisc.h` that define the
characteristics of the target CPU.

TABLE 5-4   CPU Abstraction Interface

Source File	Description
`portlibs/jit/risc/include/porting/` `jitrisc.h`	CPU abstraction interface.
`arm/javavm/include/jit/` `jitrisc_cpu.h`	ARM implementation.
`arm/javavm/include/jit/` `jitasmconstants_cpu.h`	
`linux-arm/javavm/include/jit/` `jitasmmacros_cpu.h`	

`portlibs/jit/risc/include/porting/jitrisc.h` defines the CPU
abstraction interface, including macros that describe available registers and control
facilities like condition code settings and the availability of conditional instruction
execution.

The CPU abstraction interface identifies which registers to use, allocate and avoid.
The following list describes many of the macros in
`portlibs/jit/risc/include/porting/jitrisc.h`.

- `CVMCPU_SP_REG`: register number of the C stack pointer.
- `CVMCPU_JSP_REG` and `CVMCPU_JFP_REG`: dedicated Java stack pointer and Java
  frame pointer registers. They must be in the set that are safe (non-volatile) across
  C calls.
- `CVMCPU_PROLOGUE_PREVFRAME_REG` and `CVMCPU_PROLOGUE_NEWJFP_REG`:
  registers to be used in the prolog. These are used when calling some helpers in a
  compiled method's prolog. They must be in the set that is safe across C calls. But
  they are only used during the prolog so will also be available for use by the
  compiled code.
- `CVMCPU_FIRST_PHI_SPILL_REG` and `CVMCPU_MAX_PHI_SPILLS_IN_REGS`:
  registers to be used for holding computed values between blocks. These are
  usually at the low end of the range of non-volatile registers.
- `CVMCPU_ARG1_REG`, ... `CVMCPU_ARG4_REG`, `CVMCPU_RESULT1_REG` and
  `CVMCPU_RESULT2_REG`: the C argument and return value registers. These are
  volatile across C calls.
- *Optional.*
  - `CVMCPU_CHUNKEND_REG`: Java stack chunk end pointer
  - `CVMCPU_CVMGLOBALS_REG`: system globals pointer
  - `CVMCPU_CP_REG`: constant pool pointer

- ■ `CVMCPU_EE_REG`: ee pointer
- ■ `CVMCPU_ZERO_REG`: zero register
- ■ Register manager configuration. (*Required.*)

  Note: these guidelines are for general purpose registers. There are corresponding macros for floating point registers.

  - ■ There should be no more than 32 general purpose registers numbered `0...(n-1)`, because the bitset for them must be represented with a `CVMUint32`.
  - ■ `CVMCPU_ALL_SET`: the bitset of all registers.
  - ■ `CVMCPU_BUSY_SET`: the bitset of all registers that the register manager should not allocate, and which it doesn't already know about. (It already knows about any registers named in any of the above definitions, including C stack pointer, Java frame pointer, Java stack pointer and any optional dedicated registers). For example, registers reserved for use by the OS are included in `CVMCPU_BUSY_SET`.
  - ■ `CVMCPU_NON_VOLATILE_SET`: the bitset of all registers safe across a C function call.
  - ■ `CVMCPU_VOLATILE_SET`: the bitset of all registers unsafe across a C function call.
  - ■ `CVMCPU_AVOID_SET`: the register set that can be used, but only as a last resort. Generally this will include the argument registers plus some or all of the phi registers.
  - ■ `CVMCPU_MIN_INTERESTING_REG` and `CVMCPU_MAX_INTERESTING_REG`: the lower and upper bounds on register numbers that the register manager should consider for allocation. This is used by the register manager's search optimization so that the register manager does not have to iterate over registers that can never be used.
- ■ *Optional CPU features.*
  - ■ `CVMCPU_HAS_CONDITIONAL_ALU_INSTRUCTIONS`, `CVMCPU_HAS_CONDITIONAL_LOADSTORE_INSTRUCTIONS` and `CVMCPU_HAS_CONDITIONAL_CALL_INSTRUCTIONS` indicate conditional instruction support.
  - ■ `CVMCPU_HAS_ALU_SETCC` indicates that ALU instructions can set condition codes.
  - ■ `CVMCPU_HAS_IMUL_IMMEDIATE` indicates that the platform has an integer mul by immediate instruction.
  - ■ `CVMCPU_HAS_POSTINCREMENT_STORE` indicates that the platform has a postincrement addressing mode for stores.
  - ■ `CVMCPU_MAX_LOADSTORE_OFFSET` shows the maximum offset (+/-) for a load/store word instruction.
  - ■ `CVMCPU_RESERVED_CP_REG_INSTRUCTIONS` shows the number of instructions reserved for setting up constant pool base register in the method's prologue.

- CVMCPU_NUM_OF_INSTRUCTIONS_FOR_GC_PATCH gives the number of nop's needed at GC check patch points, which is normally the case on processors with a delay slot after a branch-and-link instruction. Nop's are alse needed if the call that is patched in cannot be done in one instruction.

5.3.3 Glue Code

The glue functions in `share/javavm/include/porting/jit.h` and `portlibs/jit/risc/include/porting/ccmrisc.h` are short routines that bind compiled code to C helper functions.

TABLE 5-5   Glue Code

Source File	Description
`share/javavm/include/porting/` `jit.h`	Defines `CVMJITgoNative` and `CVMJITexitNative` glue functions.
`portlibs/jit/risc/include/porting/` `ccmrisc.h`	Defines glue functions for binding to compiled code to CCM C helper functions and for doing method invocations.
`arm/javavm/runtime/jit/` `jit_cpu.S`	Implementations of `CVMJITgoNative` and `CVMJITexitNative` glue functions.
`arm/javavm/runtime/jit/` `ccmglue_cpu.S`	Implementations of parts of `ccmrisc.h`.

5.3.4 Miscellaneous Code

Some of the code for a port falls outside of neat categories.

TABLE 5-6 Miscellaneous Code

Source Files	Description
`arm/javavm/runtime/jit/` `ccmcodecachecopy_cpu.S`	Convenience wrapper file for compiling together code for copying into the code cache and to guarantee the ordering of symbols.
`arm/javavm/runtime/jit/` `jitinit_cpu.c`	Initializing and destroying the back-end.

`linux-arm/javavm/runtime/jit/`
  `jit_arch.c`
`arm/javavm/runtime/jit/`
  `flushcache.S`
	Called by shared code. Also includes signal handler for handling trap-based `NullPointerExceptions`.
	
	Cache flushing allows data written by the compiler to be reinterpreted as instructions. This may be implemented with a system call.

5.3.4.1 Code Cache Copy

Dynamically compiled code makes many calls to various helper functions written in assembler. On most platforms these calls need to be made using a multiple instruction long call. This is because dynamically compiled code is in the malloc heap, and is normally too far away from code linked with the CDC-HI Java runtime binary to call it with a single instruction.

In order to locate the assembler functions closer to the dynamically compiled code so they can be called with a single instruction, the assembler functions can optionally be copied into the start of the code cache where the dynamically compiled code resides. This usually results in better performance.

The copying of the assembler functions is handled by shared code. It is enabled by `#define CVM_JIT_COPY_CCMCODE_TO_CODECACHE`. This will cause all code between the symbols `CVMCCMcodeCacheCopyStart` and `CVMCCMcodeCacheCopyEnd` to be copied. Normally `ccmcodecachecopy_cpu.S` is used to properly setup these two symbols.

A table of all functions that are copied must be setup in `CVMJITinitCompilerBackEnd()`, which is usually implemented in `jitinit_cpu.c`. The functions in the table must appear in the same order that they appear in memory.

Note that if you enable `CVM_JIT_COPY_CCMCODE_TO_CODECACHE`, debugger breakpoints set in the assembler code won't function properly. If they are set after running CDC, then they will never be hit since the code cache copy of the functions are the ones actually used. If they are set before running CDC, then the trap instruction inserted by the debugger will get copied into the code cache copy. This will cause the breakpoint to be hit, but the debugger won't know it's a breakpoint and get will get confused.

5.3.4.2   Trap-based `NullPointerExceptions`

The implementations of many Java bytecodes, such as `opc_invokevirtual`, need to first check if an object reference is null, and throw a `NullPointerException` if it is. Dynamically compiled methods must also do the equivalent of a null pointer check. This leads to slower performance and increased generated code.

The CDC-HI dynamic compiler allows for a more lazy approach to check for null object references called trap-based `NullPointerExceptions`. It eliminates doing an explicit check for a null object reference, followed by a conditional branch to throw the exception. Instead the dereference of the null object reference is allowed to cause a crash. This results in a `SIGSEGV` on most platforms. A signal handler must be installed to catch this signal, confirm that it occurred in compiled code, and cause execution to resume in code that will throw a `NullPointerException`.

The Linux/ARM port implements the signal handler in `jit_arch.c`. The `handleSegv()` function catches the signal, changes the link register to the instruction after the crash occurred, and changes the pc to point to the `CVMCCMruntimeThrowNullPointerExceptionGlue()` routine. This exactly mimics the compiled code calling `CVMCCMruntimeThrowNullPointerExceptionGlue()` itself from the point of the crash, which is the desired behavior when using a null object references.

Implementing trap-based `NullPointerExceptions` is entirely optional. You can choose to implement it to increase performance. To disable it, make sure you don't `#define CVMJIT_TRAP_BASED_NULL_CHECKS`. Once the port is working, you can choose to enable trap-based `NullPointerExceptions` and implement the signal handler for it.

5.3.5 Intrinsics

`java.lang` contains a few methods that are implemented with intrinsics for the dynamic compiler.

TABLE 5-7   Intrinsics

Source Files	Description
`share/javavm/include/jit` `jitintrinsic.h`	Intrinsic methods.
`portlibs/jit/risc/` `ccmintrinsics_risc.c`	Shared RISC implementation.

`arm/javavm/runtime/jit/`
  `ccmintrinsics_cpu.c`
`arm/javavm/runtime/jit/`
  `ccmintrinsics_asm_cpu.S`
	ARM implementation.

Adding intrinsics is optional but the procedure is straightforward.

1. **Create a configuration record for the intrinsic. See the example in** `arm/javavm/runtime/jit/ccmintrinsics_cpu.c`**. The record must be inside** `CVMJIT_INTRINSIC_CONFIG_BEGIN` **and** `CVMJIT_INTRINSIC_CONFIG_END` **markers. This record will include a pointer to the intrinsic glue implementation.**

2. **Override the** `CVMJITintrinsicsList` **intrinsic list in** `<`**CPU**`>/javavm/include/jit/jit_cpu.h`**.**

5.3.6 Invokers

The invoker functions handle the transitions between dynamically compiled methods and interpreted code. These functions are critical to performance and aggressive techniques are necessary, including reducing the number of memory accesses, hand scheduling the assembler code and taking into account properties of methods known at compile time.

TABLE 5-8   Invokers

Source File	Description
`share/javavm/include/porting/jit/` `   ccm.h`	Shared code portability interface for invokers.
`portlibs/jit/risc/include/porting/` `   ccmrisc.h`	RISC portability interface for invokers.
`arm/javavm/runtime/jit/` `   ccminvokers_cpu.S`	Invoker implementation.

5.3.7 Emitters

Each port requires a collection of emitters that map the IR instructions to the native encoding of the target CPU architecture. Sometimes they do more if more than one native instruction must be emitted. Writing the emitters requires detailed knowledge

of the target CPU architecture. This porting interface is well-defined but requires close attention to the target CPU's capabilities. TABLE 5-9 shows the relevant source files for both the required interface and the ARM implementation.

TABLE 5-9   Emitters

Source File	Description

portlibs/jit/risc/include/porting/
  jitriscemitter.h
share/javavm/include/porting/jit/
  jit.h
	Emitter portability interface.

arm/javavm/runtime/jit/
  jitemitter_cpu.c
arm/javavm/include/jit/
  jitriscemitter_cpu.h
arm/javavm/include/jit/
  jitriscemitterdefs_cpu.h
portlibs/jit/risc/
  jitemitter.c
	Emitter implementation.

Implementing the emitters will require a detailed understanding of the target CPU. The list below provides some guidelines for the required information.

- *Instruction set issues.*
  - The emitters use the binary native encoding for the target CPU, not an assembly language.
  - Some ISAs include instructions that are supported by traps into the OS. This would affect performance.
  - Multiply-by-immediate.
- *Branch instructions.*
  - Are delay slots required after branches?
  - Reach of branches and calls.
  - Which register(s) can be used in indirect branches?
- *Load/store addressing modes.*
  - Post ++ store instruction.
  - Reach of offsets. (assume reg+offset).
  - Mechanism for loading and storing 64-bit quantities, even if it requires multiple instructions.
  - Because of the way the Java stack is organized, memory alignment beyond word alignment cannot be guaranteed.
- *Range of ALU instruction immediates.*
- *Use of condition codes (if any).*
  - Which instructions can set the condition codes.
  - Which instructions can be conditionally executed, depending on condition codes.

- If the processor lacks condition codes (e.g. MIPS) there must be a way to compare and branch (see programming idioms below).

After implementing the emitter functions, the dynamic compiler should be operational and ready for testing. The next stage is to implement helper functions.

5.3.8 Helpers

The back-end can optimize code at runtime or use pre-optimized static code modules called *helpers* that represent complex bytecode fragments that are difficult to compile dynamically. Because the shared code contains C implementations for every helper function, this work can be performed after first getting the port working. The goal here is to identify important helper functions and reimplement them, usually in assembly code.

Note: Most helpers require a small amount of assembler glue to bind compiled code to the C helper.

TABLE 5-10 Helpers

Source File	Description

`share/javavm/include/porting/jit/`
`  ccm.h`
`portlibs/jit/risc/include/porting/`
`  ccmrisc.h`
	Helper portability and glue interface.

`share/javavm/include/`
`  ccm_runtime.h`
`share/javavm/runtime`
`  ccm_runtime.c`
	Shared C reference implementation.

`arm/javavm/include/jit/`
`  ccm_cpu.h`
`arm/javavm/runtime/jit/`
`  ccmallocators_cpu.S`
`arm/javavm/runtime/jit/`
`  ccmmath_cpu.S`
	Helper implementation.

`share/javavm/include/porting/jit/ccm.h` includes a series of macros that indicate whether the helper function is provided by the platform implementation or uses the default shared C reference implementation. The the platform implementation can override the defaults with `#defines` in `<`**CPU**`>/javavm/include/jit/ccm_cpu.h`.

The ARM implementation includes more optimizations than the other ports. The
following arithmetic and allocator helpers represent a good place to start.

```
#define CVMCCM_HAVE_PLATFORM_SPECIFIC_IDIV

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_IREM

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LMUL

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LNEG

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LSHL

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LSHR

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LUSHR

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LAND

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LOR

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LXOR

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_NEW

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_NEWARRAY

#define CVMCCM_HAVE_PLATFORM_SPECIFIC_ANEWARRAY
```

Note: you may choose to handle common cases with assembly language
implementations and defer more complex cases for the C implementation. Object
synchronization and object allocation helpers are usually implemented this way,
providing high performance in the most common cases and ease of implementation
in the more difficult cases.

5.3.9   Floating Point Support

The floating point helpers provide an opportunity to implement much faster
functions than those provided with the C runtime. This is because Java floating-
point semantics are much simpler than full IEEE floating-point semantics because
there are no modes or exceptions.

CVM_JIT_USE_FP_HARDWARE causes the dynamic compiler to produce floating point instructions. Otherwise all floating point values are stored in general purpose registers and C helpers are called to perform most floating point operations. This is the softfloat model used by the ARM implementation. The C helpers can be overridden by assembly language helpers for better performance.

TABLE 5-11   Floating Point Support

Source File	Description
share/javavm/include/porting/jit/ ccm_runtime.h	Defines the helper porting interface.
share/javavm/runtime/ ccm_runtime.c	Shared floating point helpers.
arm/javavm/runtime/jit/ ccmmath_cpu.S	Assembly language floating point helpers.

PART IV   Garbage Collector Layer

This part describes the how to create a pluggable garbage collector.

This part contains the chapters:

- Creating a Garbage Collector
- Direct Memory Interface Reference
- Indirect Memory Interface Reference
- How to be GC-Safe

Note – The garbage collection APIs are optional because the built-in garbage collection algorithms are highly optimized and known to work across a wide range of applications. These APIs are made available for product-specific needs and are not required for general porting.

Creating a Garbage Collector

This chapter describes how to create a garbage collector for the CDC HotSpot Implementation Java virtual machine. Creating a garbage collector is an **optional** part of porting the CDC HotSpot Implementation Java virtual machine to a new target system because the default generational garbage collector performs well under most circumstances.

Note – This chapter assumes basic knowledge of conventional garbage collection (GC) algorithms, such as mark-and-sweep and copying collection, as well as related GC concepts, such as read and write barriers.

The chapter covers the following topics:

- Introduction
- Exactness
- Pluggable GC
- Writing a New GC

6.1 Introduction

The CDC HotSpot Implementation Java virtual machine's memory system possesses the following features:

- **Exactness**: ensures that the GC knows about all pointers at GC time; there is no need for conservative scans of the heap.
- **Pluggable GC**: allows a GC author to write a new GC without changing the VM source.

CDC HI has a generational garbage collector as its default, resulting in much reduced average GC pause times and much reduced total time spent on GC. CDC HI incorporates an implementation of generational GC as its default GC. Generational

GC is based on the observation that most objects are short lived. In other words, "young objects die young." So the heap is separated into a small *young objects area* and a large *old objects area*. Objects are allocated in the young area, and if they survive enough GCs, they are promoted to the old area. Since most objects die young, young space collections collect most of the garbage with relatively little effort and short running time, with occasional longer-running, old space collections to collect the whole heap. In CDC HI, young space collections are performed using a "semispace copying" GC algorithm, whereas old space collections are performed using a "mark-compact" GC algorithm.

Further details of CDC HI's generational GC are beyond the scope of this document. This document instead focuses on aspects of the CDC HI memory system architecture that apply to any GC written for CDC HI.

6.2   Exactness

CDC HI is built with the goal of *exactness* in mind. An exact VM knows about all pointers to the Java heap, both from the VM code and native methods. Exactness has numerous advantages:

- Allows for lower per-object overhead through the elimination of handles
- Makes full heap compaction possible on every GC
- Eliminates unnecessary object retention due to *conservative* guesses
- Allows the implementation of the widest range of GC algorithms

CDC HI implements exactness by using *GC-safe points* (or *GC points*, for short) for GC. GC cannot occur at arbitrary points in the execution of a program, but only when all threads can tolerate GC. Furthermore, threads only make their states explicit at well-known intervals, but not all the time.

Each thread in CDC HI can be in a GC-unsafe state or a GC-safe state. A thread in a GC-unsafe state is free to point to heap objects directly and can do any GC-unsafe pointer manipulations it likes. However, such a thread cannot tolerate GC, as the collector cannot obtain its precise set of pointers. A thread in a GC-safe state must make all its pointers known to the GC. In order to prevent pointers from becoming invisible to the GC through C compiler optimizations on VM code, a thread is not allowed to point to objects directly, but only through an extra level of indirection. Also, a thread registers with the GC any pointers that the GC needs to scan. Therefore, threads can tolerate GC.

The GC can only proceed when all threads are GC-safe. CDC HI makes precise pointer information available to the GC when all threads are GC-safe.

For a guide to writing GC-safe code in CDC HI and details on CDC HI internal APIs with regards to GC-safe and GC-unsafe modes, please refer to Section 3, "" on page 3-1.

6.2.1   Global GC Requests

A typical exact GC cycle in CDC HI is initiated by a thread requesting a GC. At this point, CDC HI must bring all other threads to GC points before the GC can proceed:

- Each thread polls for a global GC request at GC points.
- Upon detecting a global GC request, a thread at a GC point saves its GC-observable state and suspends itself.
- The GC requester waits for all GC-unsafe threads to become GC-safe and suspend themselves.
- When all threads are GC-safe, the GC can proceed and scan the exact state of each thread.

GC points fall on certain byte codes to ensure that each thread can suspend itself in bounded time if there is a global GC request.

Valid GC points in the CDC HI interpreter are:

- Method invocation points
- Backwards branches
- Class loading and constant resolution points
- JNI implementation heap access points
- Memory allocation points

6.2.1.1   Method Invocation Points

Method invocation points are used as GC points because of the state of the interpreter stack when a GC occurs. Since each frame on the stack refers to a method that has stopped, naturally, at a method invocation point, it makes sense to use invocation sites as GC points. So when the GC walks the interpreter stack frames looking for roots, it can readily find frame pointers into the heap.

6.2.1.2   Backwards Branches

Backwards branches are used as GC points in the interpreter to ensure that the currently executing method of each thread is guaranteed to become GC-safe within a bounded amount of time: each method will either loop by a backwards branch or hit a method invocation within a bounded amount of time.

6.2.1.3   Class Loading and Constant Resolution Points

CDC HI code outside of the interpreter, such as the system class loader and verifier, runs mostly GC-safe, in contrast to the byte code interpreter. This allows GC to occur alongside class loading, for example.

6.2.1.4   JNI Implementation

And finally, CDC HI's implementation of the Java Native Interface (JNI) allows all native methods to run GC-safe, except when they access the Java heap. So native methods can tolerate GC until they call JNI functions that access the heap. At such heap access points, the CDC HI JNI implementation makes the caller thread temporarily GC-unsafe while it accesses the heap.

6.2.1.5   Memory Allocation Points

Heap allocation points are used as GC points in the interpreter to make sure that if an allocation causes a GC, the state of the thread that initiated the GC is scannable.

6.3   Pluggable GC

CDC HI is designed to allow a GC author to write a new GC without changing a line of the VM code itself. This section describes the internal organization of the CDC HI, including the following features:

- Memory system related functions are separate from the rest of the VM, with the interfaces between the memory system and the VM clearly defined.
- Entry points to the memory system from the VM are clearly defined. This abstracts away the details of many common GC tasks from the GC author and is available as a set of routines for the GC author to use.
- GC-algorithm-independent code is separate from the GC-algorithm-dependent code. GC-algorithm-independent code is designed to be an interface that needs to be implemented by a GC author to provide GC functionality.
- GC execution flow, including object allocation, is defined.

6.3.1   Separate Memory System

The separation of the VM from the memory interface is achieved by extensive use of internal interfaces that are built hierarchically.

The VM needs to access the heap:

- Directly for GC-unsafe code
- Indirectly for GC-safe code
- Indirectly for native method code

Direct heap access is achieved by using the direct memory interface. Indirect heap access is achieved using the indirect memory interface, which is built on top of the direct memory interface. Native method heap access is achieved through a JNI implementation that is built on top of the indirect memory interface. Consequently, all heap access in the system is guaranteed to eventually go through the direct memory interface.

To achieve a memory interface that can accommodate as many GC algorithms as possible, CDC HI allows the implementation of *read* and *write barriers*. Barriers are used to ensure consistency between a running program and the GC, especially when the GC does not handle the whole heap on every GC call. Examples of such GCs are generational, incremental, and concurrent GCs. Barrier use varies widely between GC algorithms.

A read or write barrier of the data type <T> is a GC-supplied callback to be invoked on every read or write of a heap location of type <T>.

CDC HI implements support for read and write barriers below the direct memory interface implementation, so that they are not visible to the VM author. The read and write barriers are called implicitly and automatically by the implementation of the direct memory interface, and are, therefore, incurred on all heap access in the system.

6.3.2 Entry Points to GC Code

Besides accessing the heap, the VM also needs to:

- Initialize the heap on VM startup
- Call the object allocator
- Destroy the heap on VM teardown

Heap initialization/teardown and object allocation are the most commonly used entry points to GC code from the VM. All allocation and GC activity in the system is triggered by a call from the VM or a native method into the object allocator. The object allocator encapsulates GC policy and is responsible for initiating GC when it is required.

There are other entry points that the VM uses to pass control to the GC. However, these are usually triggered by a matching native call to request GC action as described below:

- The sun.misc.GC class (responsible for asynchronous, background GC) calls into the GC to figure out the time stamp of the last major GC in the system.

- The library method `Runtime.gc()` causes a GC to occur.
- The library methods `Runtime.freeMemory()` and `Runtime.totalMemory()` obtain information from the GC regarding free and total memory sizes in the heap.

6.3.3 Shared Memory System Code

There are certain activities that all GCs will have to perform, regardless of algorithm. CDC HI separates those routines into the shared GC interface called `gc_common`. Such common GC activities include:

- Making threads stop at GC-safe points
- Finding and scanning exact roots of the system
- Finding and scanning references in heap objects and arrays
- Handling special scans for
  - Weak references and finalization
  - String interning
  - Java language synchronization data structures
  - Class unloading

The details of such activities are abstracted in the implementation of the `gc_common` interface, and are available as GC services for the GC author to use. These routines and macros are described in detail in the section, Section 6.4, "Writing a New GC" on page 6-7.

6.3.4 GC-specific Memory System Code

There are certain activities that are GC-algorithm specific. CDC HI separates those routines into a GC-implementation specific GC interface `gcimpl`. The routines and macros in this interface need to be implemented by the GC author. Such GC calls are responsible for:

- Allocating and initializing the heap and its associated data structures
- Allocating new objects
- Performing the object reclamation functions of the GC
- Implementing read and write barriers.

The `gcimpl` routines will be called by the VM at appropriate points to ensure the correct GC execution flow. These routines and macros are described in detail in the section, Section 6.4, "Writing a New GC" on page 6-7.

6.3.5 GC Execution Flow

Object allocation, and subsequent possible GC action, is initiated by the VM by calling into the `gcimpl` object allocation routine and is performed by switching back and forth between shared and GC-implementation specific code.

1. CDC HI allocates memory using the shared routine `CVMgcAllocNewInstance()`.

2. `CVMgcAllocNewInstance()` does some processing and calls the GC-specific `CVMgcimplAllocObject()` to allocate the actual space for the object.

3. `CVMgcimplAllocObject()` performs the GC and calls the shared routine `CVMgcStopTheWorldAndGC()` to stop all threads at GC-safe points.

4. `CVMgcStopTheWorldAndGC()` ensures that all threads rendezvous at GC-safe points. When that is done, it calls the GC implementation `CVMgcimplDoGC()` to perform the GC action.

5. `CVMgcimplDoGC()` may call shared GC service routines to scan GC state: For example, `CVMgcScanRoots()` to scan all roots or `CVMobjectWalkRefs()` to scan the pointers in a given object or array.

6. When `CVMgcimplDoGC()` returns, all threads that were stopped at GC points resume execution. Eventually, `CVMgcAllocNewInstance()` returns and the thread that originally initiated GC resumes execution.

6.4 Writing a New GC

To write a new garbage collector for CDC HI, the GC author implements the `gcimpl` interface. This section outlines the GC and relevant CDC HI source organization, describes the CDC HI data types that the GC author needs to know about, and describes the `gcimpl` routines that need to be implemented. This section also describes the shared GC routines available to the GC author.

6.4.1 Source Organization

The important directories and files relevant to implementing GCs on CDC HI are:

- `src/share/javavm/include/gc_common.h`
  The shared GC interface.
- `src/share/javavm/include/gc/gc_impl.h`
  The gcimpl GC interface that has to be implemented for each GC.

- `src/share/javavm/include/gc/<gcname>/gc_config.h`
  The configuration file for a specific GC.
- `src/share/javavm/runtime/gc/<gcname>/gc_impl.c`
  The implementation file for a specific GC.

6.4.2   Data Types

GC code has direct access to all objects on the heap. The GC code can also assume
that it is single-threaded if it is being executed as part of a `CVMgcimplDoGC()`,
which guarantees that no GC-unsafe threads exist. Therefore, GC code can refer to
an object directly using the type `CVMObject*`. A `CVMObject` is defined as:

```
struct CVMjava_lang_Object {
    CVMObjectHeader        hdr;
    CVMJavaVal32           fields[1];
};
...
typedef CVMjava_lang_Object CVMObject;
```

where the object header `CVMObjectHeader` is defined as:

```
struct CVMObjectHeader {
    CVMClassBlock   *clas;
    volatile CVMUint32  various32; /* A multi-purpose field */
};
```

Array types are a *subclass* of object types, in the sense that an array reference in CDC
HI can be cast to a `CVMObject*`. The header of a Java language array contains the
same structure, with an additional 32-bit length field.

Array types differ for each element type. Array elements are *tightly packed* in that an
individual sub-word element of an array, such as a `short`, is not widened to `int`
width. A sample array declaration with name <arrName> and element type
<elem_type> looks like:

```
struct CVMArrayOf<arrName> {
    CVMObjectHeader hdr;
    CVMJavaInt      length;
    <elem_type>     elems[1];
};
typedef struct CVMArrayOf<arrName> CVMArrayOf<arrName>;
```

The CDC HI array types are:

TABLE 6-1   CDC HI Array Types

Array type	Element type
CVMArrayOfByte	CVMJavaByte
CVMArrayOfShort	CVMJavaShort
CVMArrayOfChar	CVMJavaChar
CVMArrayOfBoolean	CVMJavaBoolean
CVMArrayOfInt	CVMJavaInt
CVMArrayOfRef	CVMObjectICell
CVMArrayOfFloat	CVMJavaFloat
CVMArrayOfLong	CVMTwoJavaWords
CVMArrayOfDouble	CVMTwoJavaWords

The element type of `CVMTwoJavaWords` for the `long` and `double` cases is defined as:

```
typedef CVMJavaVal32 CVMTwoJavaWords[2];
```

Any array can be cast to `CVMArrayOfAnyType` if the aim is to access array header elements only.

Because the GC can assume single-threaded execution, it is free to override the second word of an object header, assuming that it reconstructs it before threads are resumed. The second header word frequently has a *trivial*, well known default value. This word can be tested for triviality to determine if an overriding GC routine needs to save away the original contents of the word:

```
/* The default trivial contents of the various32 word */

constant CVMUint32 CVM_OBJECT_DEFAULT_VARIOUS_WORD


/* Is a various32 word trivial?
 * (i.e., can just be set to CVM_OBJECT_DEFAULT_VARIOUS_WORD after
 * GC)
*/

CVMBool CVMobjectTrivialClassWord(CVMUint32 word)
```

The complete set of operations on an object the GC author can call are given below in the section, Section 6.4.4.5, "Per-object Data" on page 6-24.

For more information about CVM-internal data structures, see the header files included in `src/share/javavm/include`. Especially important are:

- `objects.h`: Object format, basic Java language synchronization operations
- `classes.h`: CDC HI internal representations of Java language classes
- `gc_common.h`: Shared GC data structures
- `sync.h`: Lock structures

Also note that the basic data types defined by the platform and the VM basic data types that are exported to the porting layer are described in the chapter Section 3, "" on page 3-1:

- The `defs.h` section describes the data types defined by the platform.
- The `vm-defs.h` section describes data types that are exported to the porting layer.

6.4.3 What to Implement

As described previously, a new GC is written by implementing a set of `gcimpl` functions. This section describes them, detailing how to get basic GC execution, including the functions and macros that must be implemented, the read and write barriers to use, and how to move arrays.

6.4.3.1 Basic Execution

For basic GC execution, and for interfacing with the VM, the GC implementation must implement the following data types and functions.

To start out with, define in `src/share/javavm/include/gc/<gcname>/gc_config.h`:

```
struct CVMGCGlobalState {

    ....

};
```

This should include any global state the GC would like to maintain which non-GC code might wish to access. In the current state of CDC HI, there are no such details that may be communicated through `CVMGCGlobalState`. This may change in the future.

Now the GC author should implement the following functions.

1. For heap initialization:

```
/* Initialize GC global state if required */

CVMUint32 CVMgcimplInitGlobalState(CVMGCGlobalState* globalState)


/* Initialize the heap, with a given minimum and maximum heap size
```

in bytes. Return CVM_TRUE on success, CVM_FALSE otherwise. */

CVMBool **CVMgcimplInitHeap**(CVMGCGlobalState* globalState, CVMUint32
minBytes, CVMUint32 maxBytes)

2. For allocation and GC:

```
/*
 * Allocate uninitialized heap object of size numBytes
 * This is called by the VM code on every allocation.
 */
CVMObject* CVMgcimplAllocObject(CVMExecEnv* ee, CVMUint32 numBytes)


/*
 * Perform GC.
 *
 * This routine is called by the common GC code after all locks are
 * obtained, and threads are stopped at GC-safe points. It's the
 * GC routine that needs a snapshot of the world while all threads
 * arestopped (typically at least a root scan).
 *
 * The goal to free is 'numBytes' bytes.
 */
void CVMgcimplDoGC(CVMExecEnv* ee, CVMUint32 numBytes)
```

3. For teardown and VM exit:

```
/* Teardown routines */


/*
 * Destroy GC global state
 */
void CVMgcimplDestroyGlobalState(CVMGCGlobalState* globalState);


/*
 * Destroy heap. CVM_TRUE on success, CVM_FALSE otherwise.
 */
CVMBool CVMgcimplDestroyHeap(CVMGCGlobalState* globalState);
```

4. Miscellaneous routines:

```
/*
```

```
   * Return the number of bytes free in the heap.
 */
CVMUint32 CVMgcimplFreeMemory(CVMExecEnv* ee)


/*
 * Return the amount of total memory in the heap, in bytes.
 */
CVMUint32 CVMgcimplTotalMemory(CVMExecEnv* ee)


/* The time stamp of the last full GC of the heap, in order to
 * support the implementation of
 * sun.misc.GC.maxObjectInspectionAge(). This should return the
 * value of CVMtimeMillis() obtained on the last GC performed.
 */
CVMInt64 CVMgcimplTimeOfLastMajorGC();
```

5. Debug-only routines (when CVM_DEBUG=true at build-time)

```
/* Heap iteration support */


/*
 * Per-object callback to call during iteration
 */
typedef void (*CVMObjectCallbackFunc)(CVMObject* obj, CVMClassBlock*
cb, CVMUint32 objSize, void* data);
/*
 * CVMgcimplIterateHeap should traverse all objects on the heap
 * and call 'cback' on each object, with its class, size and
 * generic 'data'.
 *
 * If the heap consists of contiguous range(s), use
 * CVMgcScanObjectRange()
 */
void CVMgcimplIterateHeap(CVMExecEnv* ee, CVMObjectCallbackFunc
cback, void* data)


/*
 * A per-object callback function, to be called during heap dumps
```

```
*/
typedef void (*CVMObjectCallbackFunc)(CVMObject* obj, CVMClassBlock*
cb, CVMUint32  objSize, void* data);


/*
 * Heap dump support: Iterate over a contiguous-allocated
 * range of objects.
 */
void CVMgcScanObjectRange(CVMExecEnv* ee, CVMUint32* base, CVMUint32*
top, CVMObjectCallbackFunc callback, void* callbackData);
```

6.4.3.2   Read and Write Barriers

CDC HI allows a GC author to define read and write barriers as required by a given
GC algorithm. This is done by including a series of #define's in
src/share/javavm/include/gc/<gcname>/gc_config.h. The implementation
of the barrier for a data type <T> is called implicitly by the appropriate direct
memory layer macro corresponding to <T>; the barriers are not visible to VM
authors.

Note that in all the barriers listed below, the type Ref refers to *any* reference type,
including objects of all classes and arrays. Appropriate type checking of assignments
is done by the rest of the VM; all reference types are equal by the time they trickle
down to the barrier layer.

The default implementation of a barrier is empty. Therefore, the GC author should
only #define the barriers that he/she needs.

The read and write barriers are separated according to data type. The names are self
explanatory. The code for a read or write barrier is executed right before the actual
read or write takes place.

All barriers take as argument a pointer to the head of the object being written, as
well as the address of the slot being written to. Write barriers take an additional
argument that is the value that is being written.

Read Barriers

The read barrier for reference typed array or object slots:

```
void CVMgcimplReadBarrierRef(
  CVMObject* objRef, CVMJavaObject** fieldLoc)
```

The read barriers for non-reference types, size 32-bits or less:

```
void CVMgcimplReadBarrierByte(
  CVMObject* objRef, CVMJavaByte* fieldLoc)
void CVMgcimplReadBarrierBoolean(
  CVMObject* objRef, CVMJavaBoolean* fieldLoc)
void CVMgcimplReadBarrierShort(
  CVMObject* objRef, CVMJavaShort* fieldLoc)
void CVMgcimplReadBarrierChar(
  CVMObject* objRef, CVMJavaChar* fieldLoc)
void CVMgcimplReadBarrierInt(
  CVMObject* objRef, CVMJavaInt* fieldLoc)
void CVMgcimplReadBarrierFloat(
  CVMObject* objRef, CVMJavaFloat* fieldLoc)
```

The read barrier for 64-bit slots, for Java language `long` and `double`.

```
void CVMgcimplReadBarrier64(
  CVMObject* objRef, CVMJava32* fieldLoc)
```

Write Barriers

The write barrier for reference typed array or object slots:

```
void CVMgcimplWriteBarrierRef(
  CVMObject* objRef, CVMObject** fieldLoc, CVMObject* rhs)
```

The write barriers for non-reference types, size 32-bits or less:

```
void CVMgcimplWriteBarrierByte(
  CVMObject* objRef, CVMJavaByte* fieldLoc, CVMJavaByte rhs)
void CVMgcimplWriteBarrierBoolean(
  CVMObject* objRef, CVMJavaBoolean* fieldLoc, CVMJavaBoolean rhs)
void CVMgcimplWriteBarrierShort(
  CVMObject* objRef, CVMJavaShort* fieldLoc, CVMJavaShort rhs)
void CVMgcimplWriteBarrierChar(
  CVMObject* objRef, CVMJavaChar* fieldLoc, CVMJavaChar rhs)
void CVMgcimplWriteBarrierInt(
  CVMObject* objRef, CVMJavaInt* fieldLoc, CVMJavaInt rhs)
void CVMgcimplWriteBarrierFloat(
  CVMObject* objRef, CVMJavaFloat* fieldLoc, CVMJavaFloat rhs)
```

The write barrier for 64-bit slots, for Java language `long` and `double`.

```
void CVMgcimplWriteBarrier64(

  CVMObject* objRef, CVMJava64* fieldLoc, CVMJava64 rhsPtr)
```

Important notes:

- In the 64-bit cases of read and write barriers, the `fieldLoc` argument of a read or write barrier is a pointer to a `CVMJavaVal32` corresponding to the first 32-bit word of a 64-bit value. Likewise, in the 64-bit write barrier, `rhsPtr` is a pointer to a `CVMJavaVal32` corresponding to the first 32-bit word of the 64-bit value being written.
- The `byte`, `char` and `short` data types are widened to `int` width in regular Java objects (not in arrays), so barriers for those data types are `CVMgcimpReadBarrierInt()` and `CVMgcimplWriteBarrierInt()`, and not the shorter variants.

6.4.3.3   Moving Arrays

The barriers described above are defined on a slot-by-slot basis. On some GCs, this may prove to be inefficient when arrays need to be moved. For array moves, optional *block readers* and *block writers* may be defined. These would have to perform the read or write and batch the barriers. If a GC chooses not to override these, the memory system invokes the element-wise barriers for each element of the array move.

Any of the block operations in these sections may be overridden:

- To read from a Java language array into a C array
- To write to a Java language array from a C array
- To copy the contents of one Java language array to another Java language array of the same type

To read from a Java language array into a C array

Each of these will read `len` elements of the appropriate type from Java language array `arr`, starting from index `start`. The elements will be written into the C buffer `buf`. The action performed below should include any block barrier action required, and also the block copy itself, which is the equivalent of

```
memmove(&buf[0], arr->elems[start], len * sizeof(<jType>))
```

where `jType` is the appropriate Java language type (e.g., `CVMJavaInt` or `CVMJavaLong`).

```
void CVMgcimplArrayReadBodyByte(

  CVMJavaByte* buf, CVMArrayOfByte* arr,

  CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyBoolean(
```

```
    CVMJavaBoolean* buf, CVMArrayOfBoolean* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyShort(

    CVMJavaShort* buf, CVMArrayOfShort* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyChar(

    CVMJavaChar* buf, CVMArrayOfChar* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyInt(

    CVMJavaInt* buf, CVMArrayOfInt* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyFloat(

    CVMJavaFloat* buf, CVMArrayOfFloat* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyRef(

    CVMJavaObject** buf, CVMArrayOfRef* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyLong(

    CVMJavaVal32* buf, CVMArrayOfLong* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayReadBodyDouble(

    CVMJavaVal32* buf, CVMArrayOfDouble* arr,

    CVMUint32 start, CVMUint32 len)
```

To write to a Java language array from a C array

Each of these will write *len* elements of the appropriate type to a Java language array *arr,* starting from index *start*. The elements will be read from the C buffer *buf*. The action performed below should include any block barrier action required, and also the block copy itself, which is the equivalent of

```
memmove(arr->elems[start], &buf[0], len * sizeof(<jType>))
```

where jType is the appropriate Java language type (e.g., CVMJavaInt or CVMJavaLong).

```
void CVMgcimplArrayWriteBodyByte(

    CVMJavaByte* buf, CVMArrayOfByte* arr,

    CVMUint32 start, CVMUint32 len)

void CVMgcimplArrayWriteBodyBoolean(
```

```
  CVMJavaBoolean* buf, CVMArrayOfBoolean* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyShort**(
```
  CVMJavaShort* buf, CVMArrayOfShort* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyChar**(
```
  CVMJavaChar* buf, CVMArrayOfChar* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyInt**(
```
  CVMJavaInt* buf, CVMArrayOfInt* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyFloat**(
```
  CVMJavaFloat* buf, CVMArrayOfFloat* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyRef**(
```
  CVMJavaObject** buf, CVMArrayOfRef* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyLong**(
```
  CVMJavaVal32* buf, CVMArrayOfLong* arr,
  CVMUint32 start, CVMUint32 len)
```

void **CVMgcimplArrayWriteBodyDouble**(
```
  CVMJavaVal32* buf, CVMArrayOfDouble* arr,
  CVMUint32 start, CVMUint32 len)
```

To copy the contents of one Java language array to another Java language array of the same type

Each of these will copy `len` elements of the appropriate type from `srcArr[srcIdx, srcIdx+len)` to `dstArr[dstIdx, dstIdx+len)`. The action performed below should include any block barrier action required, and also the block copy between the arrays itself, which is the equivalent of

```
memmove(dstArr->elems[dstIdx], srcArr->elems[srcIdx], len *
sizeof(<jType>))
```

where `jType` is the appropriate Java language type (e.g., `CVMJavaInt` or `CVMJavaLong`).

void **CVMgcimplArrayCopyByte**(
```
  CVMArrayOfByte* srcArr, CVMUint32 srcIdx,
```

```
  CVMArrayOfByte* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyBoolean(
  CVMArrayOfBoolean* srcArr, CVMUint32 srcIdx,
  CVMArrayOfBoolean* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyShort(
  CVMArrayOfShort* srcArr, CVMUint32 srcIdx,
  CVMArrayOfShort* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyChar(
  CVMArrayOfChar* srcArr, CVMUint32 srcIdx,
  CVMArrayOfChar* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyInt(
  CVMArrayOfInt* srcArr, CVMUint32 srcIdx,
  CVMArrayOfInt* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyFloat(
  CVMArrayOfFloat* srcArr, CVMUint32 srcIdx,
  CVMArrayOfFloat* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyRef(
  CVMArrayOfRef* srcArr, CVMUint32 srcIdx,
  CVMArrayOfRef* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyLong(
  CVMArrayOfLong* srcArr, CVMUint32 srcIdx,
  CVMArrayOfLong* dstArr, CVMUint32 dstIdx, CVMUint32 len)
void CVMgcimplArrayCopyDouble(
  CVMArrayOfDouble* srcArr, CVMUint32 srcIdx,
  CVMArrayOfDouble* dstArr, CVMUint32 dstIdx, CVMUint32 len)
```

6.4.4   What to Call

In the section, Section 6.3.3, "Shared Memory System Code" on page 6-6, we have mentioned the gc_common interface. This section outlines the various components of the gc_common interface available to the GC author, including how to initiate a GC, how GC roots are scanned, how special root scans are performed, how object walking is performed, and a list of the macros for accessing the data on an object. You can find the interface in src/share/javavm/include/gc_common.h.

6.4.4.1 Initiating a GC

When an object allocator decides to GC (most probably due to an allocation failure), it has to make sure that the system is stopped in a GC-safe way. In CDC HI, this is accomplished by using `CVMgcStopTheWorldAndGC()`:

```
/*
 * Initiate a GC. Acquire all GC locks, stop all threads, and then
 * call back to the particular GC to do the work. When the
 * particular GC is done, resume.
 *
 * Returns CVM_TRUE on success, CVM_FALSE if GC could not be run.
 */
CVMBool CVMgcStopTheWorldAndGC(CVMExecEnv* ee, CVMUint32 numBytes)
```

This function stops the system in a GC-consistent way by acquiring all system locks, and bringing all threads to GC-safe points. Then it calls the entry point to the GC implementation, `CVMgcimplDoGC()` to do the actual work. If GC work could not be performed because of a problem such as an out of memory situation, `CVMgcStopTheWorldAndGC()` returns `CVM_FALSE`.

6.4.4.2 Root Scans

When all threads are stopped at GC-safe points, the GC will need to scan all GC roots. This is accomplished by using `CVMgcScanRoots()`, and a GC-specific callback function:

```
/*
 * Scan the root set of collection
 */
void CVMgcScanRoots(CVMExecEnv* ee, CVMGCOptions* gcOpts,
CVMRefCallbackFunc callback, void* data)
```

where the callback function type is defined as:

```
/*
 * A 'ref callback' called on each *non-NULL* discovered root
 */
typedef void (*CVMRefCallbackFunc)(CVMObject** refAddr, void* data)
```

So the GC author defines a callback function that takes the address of a reference containing slot as argument, along with an opaque `data` argument. `CVMgcScanRoots(ee, gcOpts, refCallback, refCallbackData)` calls `(*refCallback)(refPtr, refCallbackData)` on every discovered root address `refPtr`.

The memory system guarantees that both the following conditions hold when the callback routine is called:

```
refPtr  != NULL
*refPtr != NULL
```

The roots scanned are JNI local and global references, Java language stack locations, Java language local variables, Java language static variables, and CDC HI-internal data structures. The details are abstracted from the GC author.

Weak references are only discovered and queued up if `gcOpts.discoverWeakReferences` is `CVM_TRUE` before a call to `CVMgcScanRoots()`. So the GC author typically calls the first root scan with weak references discovery enabled, and then disable weak references discovery by setting `gcOpts.discoverWeakReferences` to `CVM_FALSE`.

The root scanning operation may be performed multiple times on each GC cycle. However, note that each root scan cycle including the first one should be preceded by a call to `CVMgcClearClassMarks()`:

```
/*
 * Clear the class marks for dynamically loaded classes.
 */
void CVMgcClearClassMarks(CVMExecEnv* ee, CVMGCOptions* gcOpts);
```

This function is responsible for clearing the mark bits on dynamically loaded classes. The mark bits are used to prevent infinite recursion and redundant work on class scanning. If they are not cleared between successive root scans, the GC might end up skipping important class roots like Java language statics.

Note that `CVMgcClearClassMarks()` is a separate function since a root scan cycle may include more than just calling `CVMgcScanRoots()`. For example, a generational GC may call `CVMgcScanRoots()` to discover system roots, `CVMgcScanSpecial()` to discover special object roots, and then scan pointers recorded by a write barrier. So `CVMgcClearClassMarks()` should be called before each such list of root scans, during which class scanning state should be kept.

6.4.4.3   Special Root Scans

The Java 2 Platform language and libraries have features that require special scanning support from CDC HI. In particular, the following requires special handling:

■ Weak references and finalization

Java 2 Platform has three flavors of public weak reference classes in the `java.lang.ref` package, a fourth package-private `java.lang.ref` weak reference flavor for implementing finalization, and a JNI weak reference flavor for native method use. The GC needs to be aware of all these.

■ **String interning**

The CDC HI classloader interns Java language strings that it finds in newly loaded class files. Also, an application can intern strings using `String.intern()`. Some special treatment is necessary to enable unloading of these strings when they are not in application use.

■ **Java language synchronization data structures**

The CDC HI runtime uses various data structures to support Java language synchronization, which contain embedded pointers to objects. These data structures should be collectible when the objects they refer to become garbage.

■ **Class unloading**

According to Java 2 Platform semantics, a class *C* can be unloaded if and only if its classloader *CL* is garbage-collected. The CDC HI memory system should implement this behavior.

The CDC HI memory system hides the GC-scanning details of these special features from the GC author by the use of a narrow interface consisting of two functions. If the GC author calls these two functions at the right points in GC code, all special scanning is performed automatically.

The idea with special scanning is to garbage collect entries from out-of-heap tables pointing into the heap. If we declared the tables as GC roots, they would automatically be kept alive and their entries would never be collected. Instead, we do the special objects scanning in conjunction with information from the GC to figure out which entries of the special objects may be discarded and which entries need to be kept.

Where to insert special root scan APIs

From the CDC HI point of view, GC involves two conceptual points where special root scan APIs should be inserted:

1. **A point at which object liveness detection is possible. See details below on Section 3., "A point at which object liveness detection is possible." on page 6-22**.

2. **A point at which object pointer update is possible, from a pre-GC pointer to a post-GC pointer. See details below on Section 4., "A point at which object pointer update is possible, from a pre-GC pointer to a post-GC pointer." on page 6-23**.

As illustrated below, when the GC calls
`CVMgcProcessSpecialWithLivenessInfo()`, each special object slot will be
checked by `(*isLive)()` to see if the GC determined it to be live in the preceding
reachability scan. Dead entries will automatically be removed from the special object
tables. Later on, the GC will call `CVMgcScanSpecial()`, which will cause all
remaining live entries in the special object tables to be updated with new pointers.

3. **A point at which object liveness detection is possible.**

For a mark-sweep-compact type collector, this would be right after a recursive mark
has been performed, but before the addresses of any objects have changed. So to
detect liveness, the mark-sweep collector would simply inspect the mark of an
object. For a copying collector, this would be right after all live data has been copied.
So to detect liveness, the copying collector would check whether an object reference
is to a forwarded old-space object, or a new-space object.

For this point, the appropriate special scan routine is
`CVMgcProcessSpecialWithLivenessInfo()`.

The full API for point #1:

```
/*
 * Process special objects with liveness info from a particular GC
 * implementation. This covers special scans like string intern
 * table, weak references and monitor structures.
 *
 * isLive - a predicate that returns true if an object is strongly
 * referenced
 *
 * transitiveScanner - a callback that marks an object
 * and all its children
 */
void CVMgcProcessSpecialWithLivenessInfo(CVMExecEnv* ee,
CVMGCOptions* gcOpts, CVMRefLivenessQueryFunc isLive, void*
isLiveData, CVMRefCallbackFunc transitiveScanner, void*
transitiveScannerData);
```

The `transitiveScanner` callback function type `CVMRefCallbackFunc` is defined
in Section 6.4.4.2, "Root Scans" on page 6-19. The `transitiveScanner` should
mark its parameter object reference and all its children. The non-NULL
`CVMRefCallbackFunc` argument semantics hold for `transitiveScanner` (see
Section 6.4.4.2, "Root Scans" on page 6-19).

The liveness test is done using a predicate `isLive` of type:

```
/*
```

```
 * A predicate to test liveness of a given reference
 */
typedef CVMBool (*CVMRefLivenessQueryFunc)(CVMObject** refAddr,
void* data)
```

Here also, the non-NULL argument semantics hold for isLive (see Section 6.4.4.2, "Root Scans" on page 6-19).

4. **A point at which object pointer update is possible, from a pre-GC pointer to a post-GC pointer.**

This varies widely for each algorithm. For a copying type algorithm, objects typically include their forwarding addresses. So pointer updating for an old space pointer is simply obtaining its forwarding address. Pointer updating for a new space pointer is not necessary.

For this point, the appropriate special scan routine is CVMgcScanSpecial().

The full API for point #2:

```
/*
 * Scan and optionally update special objects. This covers special
 * scans like string intern table, weak references and monitor
 * structures.
 */
void CVMgcScanSpecial(CVMExecEnv* ee, CVMGCOptions* gcOpts,
CVMRefCallbackFunc updateRefCallback, void* data);
```

The updateRefCallback function type CVMRefCallbackFunc is defined in Section 6.4.4.2, "Root Scans" on page 6-19. It is called for each location that needs to be updated to a new address. The non-NULL CVMRefCallbackFunc argument semantics hold for updateRefCallback (see Section 6.4.4.2, "Root Scans" on page 6-19).

6.4.4.4 Object Walking

Given an object reference, the GC must be able to find all pointers embedded in the object and perform an action on each pointer. This operation is very common in all tracing GCs.

In CDC HI, object walking is performed using a macro, for maximum efficiency. The object walker uniformly and automatically handles arrays of references and objects. When it encounters an object of class *C*, it scans class *C* as well. It also discovers weak references, and acts accordingly.

```
macro void CVMobjectWalkRefsWithSpecialHandling(
```

```
CVMExecEnv* ee, CVMGCOptions* gcOpts, CVMObject* obj, CVMUint32
firstHeaderWord, C-statement refAction, CVMRefCallbackFunc callback,
void* data)
```

So given an ee and a gcOpts, CVMobjectWalkRefsWithSpecialHandling()
scans object obj with the first header word firstHeaderWord. It *executes* (more
like "substitutes", given that this is a macro) the statement refAction on every
embedded object reference. refPtr is a special variable within the body of
refAction, pointing to the object slot being scanned. The object slot may contain
NULL, so refAction must be prepared to deal with that.

CVMobjectWalkRefsWithSpecialHandling() also calls
(*callback)(refPtr, data) on the address refPtr of every reference-typed
slot in the class data. The non-NULL CVMRefCallbackFunc argument semantics
hold for callback (see Section 6.4.4.2, "Root Scans" on page 6-19).

Note that refAction is a macro, whereas callback is a function. This asymmetry
is intentional for efficiency reasons: refAction is going to be called for each slot of
each object, whereas callback is going to be called for each slot of each class. The
former is typically orders of magnitude more frequent than the latter.

6.4.4.5   Per-object Data

The following set of macros are responsible for accessing the information on an
object given a direct object reference. The set of operations are separated into a few
distinct categories. Arrays and objects automatically receive separate treatment.

■ **Object size support, given an object or array reference**

```
/* Get the size of an object or array instance */

CVMUint32 CVMobjectSize(CVMObject* obj)

/* Get the size of an object or array instance given its class*/

CVMUint32 CVMobjectSizeGivenClass(CVMObject* obj, CVMClassBlock* cb)
```

■ **Object class support**

```
/* Get the first word of the object header as integer */

CVMUint32 CVMobjectGetClassWord(CVMObject* obj)

/* Set the first word of the object header as integer */

void CVMobjectSetClassWord(CVMObject* obj, CVMUint32 word)

/* Get the class of a given object */

void CVMobjectGetClass(CVMObject* obj)
```

■ **ROMized object support**

```
/* CVM_TRUE if an object is read-only and not on the heap */

CVMBool CVMobjectIsInROM(CVMObject* obj)
```

■ **Object marking support**

```
/* CVM_TRUE if an object is "marked". */
CVMBool CVMobjectMarked(CVMObject* obj)
/* Clear mark on an object */
void CVMobjectClearMarked(CVMObject* obj)
/* Set mark on an object */
void CVMobjectSetMarked(CVMObject* obj)
```

6.4.5   Example GC

This section outlines a very simple allocator and garbage collector written for CDC HI called `markcompact`, for a mark-sweep-compact collector. The outline here only pertains to implementation for the GC interface, so there is no detail about the actual compaction process.

To begin, there should be a `gc_config.h` file describing the GC:

```
src/share/javavm/include/gc/markcompact/gc_config.h:
#ifndef _INCLUDED_MARKCOMPACT_GC_CONFIG_H
#define _INCLUDED_MARKCOMPACT_GC_CONFIG_H
#include "javavm/include/gc/gc_impl.h"
/*
 * The following header could include any mark-compact specific
 * declarations.
 */
#include "javavm/include/gc/markcompact/markcompact.h"
/*
 * Barriers in this implementation
 */
#define CVMgcimplWriteBarrierRef(directObj, slotAddr, rhsValue
/*
 * Do nothing. Just an anxample.
 */


/*
 * Global state specific to GC
 */
struct CVMGCGlobalState {
```

```
/* Nothing here */
};
#endif /* _INCLUDED_MARKCOMPACT_GC_CONFIG_H */
```

Next, there should be a `gc_impl.c` file implementing the garbage collector. Note that setting the build option `CVM_GCCHOICE=markcompact` would build this garbage collector automatically.

`src/share/javavm/runtime/gc/markcompact/gc_impl.c`:

```
#include "javavm/include/defs.h"
#include "javavm/include/objects.h"
#include "javavm/include/classes.h"
#include "javavm/include/directmem.h"
/*
 * This file is generated from the GC choice given at build
 *time.
 */
#include "generated/javavm/include/gc_config.h"
/* The shared GC interface */
#include "javavm/include/gc_common.h"
/* And the specific GC interface */
#include "javavm/include/gc/gc_impl.h"
/* The main allocation entry point */
/*
 * Allocate uninitialized heap object of size numBytes.
 * GC "policy" encapsulated here.
 */
CVMObject*
CVMgcimplAllocObject(CVMExecEnv* ee, CVMUint32 numBytes)
{
    CVMObject* allocatedObj;
    /* Actual allocation detail hidden here */
    allocatedObj = tryAlloc(numBytes);
    if (allocatedObj == NULL) {
        /* GC and re-try allocation */
        if (CVMgcStopTheWorldAndGC(ee, numBytes)) {
            /* re-try if GC occurred */
            allocatedObj = tryAlloc(numBytes);
```

```
            }
        }
        return allocatedObj;
}
/*
 * The main GC point, which CVM calls after ensuring GC-
 * safety of all threads.
 *
 * This is a mark-sweep-compact GC, with most details of
 * the sweep and compaction hidden.
 *
 * The GC uses three callback functions. These are detailed
 * below, after CVMgcimplDoGC().
 */
void
CVMgcimplDoGC(CVMExecEnv* ee, CVMUint32 numBytes)
{
    CVMGCOptions gcOpts;
    /* Set default GC options */
    gcOpts.fullGC = CVM_TRUE;
    gcOpts.allClassesAreRoots = CVM_FALSE;
    /*
     * The mark phase includes discovering weak references
    /*
    gcOpts.discoverWeakReferences = CVM_TRUE;
    /*
     * Scan all roots. markTransitively will mark a root
     * and all its "children". Its 'data' argument is the
     * GC options. A more complicated callback could pass
     * a pointer to a struct into the callback function.
    */
    CVMgcScanRoots(ee, &gcOpts, markTransitively, &gcOpts);
    /*
     * Don't discover any more weak references.
    */
    gcOpts.discoverWeakReferences = CVM_FALSE;
```

```
    /*
    * At this point, we know which objects are live and
    * which are not. Do the special objects processing.
*/
    CVMgcProcessSpecialWithLivenessInfo(ee, &gcOpts, isLive,
NULL,
   markTransitively, &gcOpts);
    /* The sweep phase. This phase computes the new
    * addresses of each object and writes them on the * side.
    * Details hidden.
*/
    sweep();
    /* Update the roots again, by writing out looking up
    * old -> new address translations.
*/
    CVMgcScanRoots(ee, &gcOpts, updateRoot, NULL);
    CVMgcScanSpecial(ee, gcOpts, updateRoot, NULL);
    /* And update all interior pointers. Details hidden */
    scanObjectsInHeap(ee, gcOpts, updateRoot, NULL);
    /* Finally we can move objects, and reset object marks.
    * Compaction details hidden.
*/
    compact();
}
/*
 * The liveness predicate, for use in special objects
 * scanning.
 */
static CVMBool
isLive(CVMObject** refPtr, void* data)
{
    CVMObject* ref;
    CVMassert(refPtr != NULL);
    ref = *refPtr;
    CVMassert(ref != NULL);
    /*
```

```
        * ROM objects are always live
      */
      if (CVMobjectIsInROM(ref)) {
      return CVM_TRUE;
      }
      /* Is object marked? It's live then. */
      return CVMobjectMarked(ref);
}
/*
 * The transitive object marker. Marks a given object and
 * all its"children".
 * This is recursive, for simplicity. A production GC
 * should really not be recursive.
 */
static void
markTransitively(CVMObject** refPointer, void* data)
{
    CVMGCOptions* gcOpts = (CVMGCOptions*)data;
    CVMObject* ref = *refPointer;
    CVMClassBlock* refCb = CVMobjectGetClass(ref);
    /*
     * ROM objects are always live
    */
    if (CVMobjectIsInROM(ref)) {
        return;
    }
    CVMobjectSetMarked(ref);
    /*
     * Now handle all the "children".
    */
    CVMobjectWalkRefsWithSpecialHandling(CVMgetEE(), gcOpts,
ref, refCb, {
        CVMObject* thisRef = *refPtr;
        if (thisRef != NULL) {
            if (!CVMobjectMarked(thisRef)) {
                markTransitively(refPtr);
```

```
                }
            }
        }, markTransitively, data);
    }
    /*
     * Update a root with the new address of an object
    */
    static void
    CVMgenMarkCompactFilteredUpdateRoot(CVMObject** refPtr, void* data)
    {
        CVMObject* ref = *refPtr;
        /*
         * ROM objects are not on the heap
        */
        if (CVMobjectIsInROM(ref)) {
            return;
        }
        *refPtr = lookupNewAddress(ref); /* Details hidden. Update
    root. */
    }
```

Direct Memory Interface Reference

This chapter describes the Direct Memory Interface functions used by the CDC HI garbage collector (GC). It covers the following topics:

- Introduction
- Object Field Accesses
- Array Accesses
- GC-safety of Threads

7.1 Introduction

These functions are used for accessing object and array fields, and also handle GC-safe points. Please refer Section , "How to be GC-Safe" on page 9-1 for context and examples.

Caution – The use of the direct heap access operations is *GC-unsafe*. Therefore, these operations should be used with extreme care in a few selected places, and only within GC-unsafe regions. VM code should use the indirect memory layer almost all the time when outside the interpreter.

7.2 Object Field Accesses

The following macros access object fields. The result-producing ones take an l-value as the last argument, and assign to it.

7.2.1   Accessing Fields of 32-bit Width

These macros use the following parameters:

- the first parameter is a direct object reference
- the second parameter is an offset in number of words from the beginning of the object, *counting the first header word as 0*
- the third parameter is an l-value to read into or a value to write

---

Note – The implementation of GC read and write barriers are hidden beneath the Ref typed accessors.

7.2.1.1   Weakly-Typed 32-bit Read and Write

macro void **CVMD_fieldRead32**(CVMObject* o, CVMUint32 off, CVMJavaVal32 res)

macro void **CVMD_fieldWrite32**(CVMObject* o, CVMUint32 off, CVMJavaVal32 res)

7.2.1.2   Strongly-Typed 32-bit Read and Write

macro void **CVMD_fieldReadRef**(CVMObject* o, CVMUint32 off, CVMObject* item)

macro void **CVMD_fieldWriteRef**(CVMObject* o, CVMUint32 off, CVMObject* item)

macro void **CVMD_fieldReadInt**(CVMObject* o, CVMUint32 off, CVMJavaInt item)

macro void **CVMD_fieldWriteInt**(CVMObject* o, CVMUint32 off, CVMJavaInt item)

macro void **CVMD_fieldReadFloat**(CVMObject* o, CVMUint32 off, CVMJavaFloat item)

macro void **CVMD_fieldWriteFloat**(CVMObject* o, CVMUint32 off, CVMJavaFloat item)

7.2.2   Accessing Fields of 64-bit Width

These macros use the following parameters:

- the first parameter is a direct object reference

- the second parameter is an offset in number of words from the beginning of the object, *counting the first header word as 0*
- the third parameter is an l-value of type `CVMJavaVal64` to read into or a value to write

The weakly-typed versions read from and write into a word-aligned two-word area pointed to by `location`.

7.2.2.1   Weakly-Typed 64-bit Read and Write

macro void **CVMD_fieldRead64**(CVMObject* o, CVMUint32 off, CVMJavaVal32* location)

macro void **CVMD_fieldWrite64**(CVMObject* o, CVMUint32 off, CVMJavaVal32* location)

7.2.2.2   Strongly-Typed 64-bit Read and Write

macro void **CVMD_fieldReadLong**(CVMObject* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMD_fieldWriteLong**(CVMObject* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMD_fieldReadDouble**(CVMObject* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMD_fieldWriteDouble**(CVMObject* o, CVMUint32 off, CVMJavaVal64 val64)

7.3   Array Accesses

The following macros access object fields. The result producing ones take an l-value as the last argument, and assign to it

7.3.1   Accessing Elements of 32-bit Width and Below

These macros use the following parameters:

- the first parameter is a direct array reference
- the second parameter is the array index *where the first array element is at index 0*
- the third parameter is an l-value to read into, or a value to write

These macros are all strongly typed. All the Java basic types are represented.

Note – The implementation of GC read and write barriers are hidden beneath the Ref typed accessors.

---

macro void **CVMD_arrayReadRef**(CVMArrayOfRef* arr,   CVMUint32 index, CVMObject* item)

macro void **CVMD_arrayWriteRef**(CVMArrayOfRef* arr,   CVMUint32 index, CVMObject* item)

macro void **CVMD_arrayReadInt**(CVMArrayOfInt* arr,   CVMUint32 index, CVMJavaInt item)

macro void **CVMD_arrayWriteInt**(CVMArrayOfInt* arr,   CVMUint32 index, CVMJavaInt item)

macro void **CVMD_arrayReadByte**(CVMArrayOfByte* arr, CVMUint32 index, CVMJavaByte item)

macro void **CVMD_arrayWriteByte**(CVMArrayOfByte* arr, CVMUint32 index, CVMJavaByte item)

macro void **CVMD_arrayReadBool**(CVMArrayOfBool* arr, CVMUint32 index, CVMJavaBool item)

macro void **CVMD_arrayWriteBool**(CVMArrayOfBool* arr, CVMUint32 index, CVMJavaBool item)

macro void **CVMD_arrayReadShort**(CVMArrayOfShort* arr, CVMUint32 index, CVMJavaShort item)

macro void **CVMD_arrayWriteShort**(CVMArrayOfShort* arr, CVMUint32 index, CVMJavaShort item)

macro void **CVMD_arrayReadChar**(CVMArrayOfChar* arr, CVMUint32 index, CVMJavaChar item)

macro void **CVMD_arrayWriteChar**(CVMArrayOfChar* arr, CVMUint32 index, CVMJavaChar item)

macro void **CVMD_arrayReadFloat**(CVMArrayOfFloat* arr, CVMUint32 index, CVMJavaFloat item)

macro void **CVMD_arrayWriteFloat**(CVMArrayOfFloat* arr, CVMUint32 index, CVMJavaFloat item)

7.3.2　Accessing Elements of 64-bit Width

These macros use the following parameters:

- the first parameter is a direct array reference
- the second parameter is the array index *where the first array element is at index 0*
- the third parameters an l-value of type `CVMJavaVal64` to read into or a value to write.

7.3.2.1　Weakly-Typed Versions

The weakly-typed versions read from and write to a word-aligned two-word area pointed to by `location`:

macro void **CVMD_arrayRead64**(<CVMArrayOf64>* o, CVMUint32 off, CVMJavaVal32* location)

macro void **CVMD_arrayWrite64**(<CVMArrayOf64>* o, CVMUint32 off, CVMJavaVal32* location)

where <CVMArrayOf64> is either CVMArrayOfLong or CVMArrayOfDouble.

7.3.2.2　Strongly-Typed Versions

macro void **CVMD_arrayReadLong**(<CVMArrayOf64>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMD_arrayWriteLong**(<CVMArrayOf64>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMD_arrayReadDouble**(<CVMArrayOf64>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMD_arrayWriteDouble**(<CVMArrayOf64>* o, CVMUint32 index, CVMJavaVal64 val64)

where <CVMArrayOf64> is either CVMArrayOfLong or CVMArrayOfDouble.

7.3.3　Miscellaneous Array Operations

All generic object operations apply to arrays as well. In particular, the header of an array object starts out with an object header that has an additional length entry, so any operation on the header of an object may be performed on an array header.

Below is an array-specific operation.

macro CVMJavaInt32 **CVMD_arrayGetLength**(<CVMArrayOfAny>* o)

where <CVMArrayOfAny> is any direct array reference.

7.4 GC-safety of Threads

Each thread has a GC-safety state associated with it. Threads that cannot *tolerate* GC are marked as GC-unsafe. A thread whose state can be scanned by GC is marked GC-safe. GC can occur only when all threads in the system are GC-safe. When a thread requests GC, all threads that are currently GC-unsafe are rolled forward to their next GC-safe point.

The following operations show how to create a GC-unsafe window of operation and how to request a GC-safe point.

7.4.1 GC-unsafe Blocks

When a GC-safe thread wants to perform a GC-unsafe operation, it marks itself as unable to tolerate GC, performs the GC-unsafe operation, and then marks itself again as GC-safe. Use CVMD_gcUnsafeExec() to create such a window of GC-unsafety. At the end of the GC-unsafe window, the thread calling CVMD_gcUnsafeExec() polls for a GC request. If there is one, the thread suspends itself to rendezvous with all the other threads rolling forward to their GC points. Execution continues after GC.

macro void **CVMD_gcUnsafeExec**(CVMExecEnv* ee, *code* gcUnsafeAction)

where ee is a pointer to the current thread's execution environment and gcUnsafeAction is a segment of GC-unsafe code.

7.4.2 GC-safe Blocks: Requesting a GC-Safe Point

GC-unsafe code must occasionally *offer* to become GC-safe to bound the time from a GC request to the beginning of GC. CVMD_gcSafeExec and CVMD_gcSafeCheckPoint are the two macros that allow that.

The CVMD_gcSafeCheckPoint macro is used for code that will not block. The assumption here is that there is some cached state in the GC-unsafe code which needs to be saved if GC is needed.

macro void **CVMD_gcSafeCheckPoint**(CVMExecEnv* ee, *code* saveAction, *code* restoreAction)

The thread calling CVMD_gcSafeCheckPoint() checks whether there is a GC request. If there is, the thread executes saveAction to save state necessary for GC, marks itself as GC-safe, and suspends itself to rendezvous with all the other threads rolling forward to their GC points. After GC completes, the thread is resumed, marks itself as GC-unsafe again, and executes restoreAction to do any caching operations necessary to continue execution.

The CVMD_gcSafeExec macro is used for code that may potentially block. In this case, whatever cached state the GC-unsafe code has must be saved before calling the macro.

macro void **CVMD_gcSafeExec**(CVMExecEnv* ee, *code* safeAction)

The thread calling CVMD_gcSafeExec() marks itself GC-safe and checks to see if there is a GC-request. If there is one, the thread suspends itself to rendezvous with all the other threads rolling forward to their GC points. After GC is over, the thread is resumed, still in a GC-safe state. It executes safeAction, potentially blocking. After waking up from the blocking action, the thread marks itself as GC-unsafe and continues with GC-unsafe execution.

Indirect Memory Interface Reference

This chapter contains the references for the indirect memory layer functions, which are used by the garbage collector (GC). It covers the following topics:

- Introduction
- ICell Manipulations
- Registered Indirection Cells
- Object Field Accesses
- Array Accesses

8.1 Introduction

These functions are used for accessing the garbage-collected heap *indirectly* using pointers to ICells. Please refer to Chapter " for context and examples.

Note – The indirect memory interface is *GC-safe*, and is appropriate for use in VM code outside of the interpreter, like the class loader, or JNI implementation.

8.2 ICell Manipulations

The *referent* of an ICell is the direct object reference encapsulated by the ICell. The indirect memory interface allows GC-safe assignments and comparisons between referents of ICells using the following macros.

1. Assign referent of `srcICellPtr` to be the referent of `dstICellPtr`:

```
macro void CVMID_icellAssign(CVMExecEnv* ee, CVMObjectICell*
destICellPtr,CVMObjectICell* srcICellPtr)
```

2. Null out the referent of `icellPtr`:

```
macro void CVMID_icellSetNull(CVMExecEnv* ee, CVMObjectICell*
icellPtr)
```

3. Test whether the referent of `icellPtr` is null, The result is in `res`:

```
macro void CVMID_icellIsNull(CVMExecEnv* ee, CVMObjectICell*
icellPtr, CVMBool res)
```

4. Test whether the referent of `icellPtr1` is the same object reference as
   `icellPtr2`. The result is in `res`:

```
macro void CVMID_icellSameObject(CVMExecEnv* ee, CVMObjectICell*
icellPtr1,CVMObjectICell* icellPtr2, CVMBool res)
```

8.3 Registered Indirection Cells

An indirection cell must be registered with GC to be scanned as a root. A registered
indirection cell is either a local root or global root. In both cases, the client code is
handed a pointer to an ICell that is part of the root set of GC.

8.3.1 Local Roots

Local roots are allocated within *local root blocks*. At the end of a local root block, all
allocated local roots within the block are destroyed. Note that local root blocks may
be nested arbitrarily.

To begin a local root block, with respect to the current thread's execution
environment, `ee`:

```
macro CVMID_localrootBegin(CVMExecEnv* ee)
```

To allocate a local root and return a pointer to it:

```
macro CVMID_localrootDeclare(<ICell>, var)
```

Here `<ICell>` may be any legal object or array ICell type.
`CVMID_localrootDeclare` allocates a new local root, nulls out its referent,
declares an `<ICell>*` `var`, and makes `var` point to the new local root. `var` is
visible within the scope of the local root block.

To end the local root block, and to discard all local roots allocated within the block:

```
macro CVMID_localrootEnd()
```

8.3.2 Global Roots

Global roots are not bound to any one thread, but are shared between all threads. They are analogous to JNI global refs. Note that unlike local roots, global roots may be allocated out-of-order (i.e., the global roots are not guaranteed to be allocated sequentially in memory).

To allocate a new global root, null out its referent and return a pointer to it:

CVMObjectICell* **CVMID_getGlobalRoot**()

To free the global root pointed to by globalRoot:

void **CVMID_freeGlobalRoot**(CVMObjectICell* globalRoot)

8.4 Object Field Accesses

The following macros access object fields. The result producing ones take an l-value as the last argument, and assign to it.

8.4.1 Accessing Fields of 32-bit Width

These macros use the following parameters:

■ the first parameter is a pointer to the execution environment (CVMExecEnv) of the current thread
■ the second parameter is a registered indirect object reference (CVMObjectICell*)
■ the third parameter is an offset in number of words from the beginning of the object, *counting the first header word as 0*
■ the fourth parameter is an l-value to read into or a value to write

---

Note – The implementation of GC read and write barriers are hidden beneath the Ref typed accessors.

8.4.1.1 Weakly-Typed 32-bit Read and Write

macro void **CVMID_fieldRead32**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal32 res)

macro void **CVMID_fieldWrite32**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal32 res)

8.4.1.2   Strongly-Typed 32-bit Read and Write

macro void **CVMID_fieldReadRef**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMObjectICell* item)

macro void **CVMID_fieldWriteRef**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMObjectICell* item)

macro void **CVMID_fieldReadInt**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaInt item)

macro void **CVMID_fieldWriteInt**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaInt item)

macro void **CVMID_fieldReadFloat**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaFloat item)

macro void **CVMID_fieldWriteFloat**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaFloat item)

8.4.2   Accessing Fields of 64-bit Width

These macros use the following parameters:

- the first parameter is a pointer to the execution environment (CVMExecEnv) of the current thread
- the second parameter is a registered indirect object reference (CVMObjectICell*)
- the third parameter is an offset in number of words from the beginning of the object, *counting the first header word as 0*
- the fourth parameter is an l-value of type CVMJavaVal64 to read into or a value to write

The weakly-typed versions read from and write into a word-aligned two-word area pointed to by location.

8.4.2.1   Weakly-Typed 64-bit Read and Write

macro void **CVMID_fieldRead64**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal32* location)

macro void **CVMID_fieldWrite64**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal32* location)

8.4.2.2 Strongly-Typed 64-bit Read and Write

macro void **CVMID_fieldReadLong**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMID_fieldWriteLong**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMID_fieldReadDouble**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal64 val64)

macro void **CVMID_fieldWriteDouble**(CVMExecEnv* ee, CVMObjectICell* o, CVMUint32 off, CVMJavaVal64 val64)

8.5 Array Accesses

The following macros access object arrays. The result-producing ones take an l-value as the last argument, and assign to it.

8.5.1 Accessing Elements of 32-bit Width and Below

These macros use the following parameters:

- the first parameter is a pointer to the execution environment (CVMExecEnv) of the current thread
- the second parameter is a registered indirect array reference (CVMArrayOf<T>ICell*)
- the third parameter is the array index *where the first array element is at index 0*
- the fourth parameter is an l-value to read into, or a value to write

These macros are all strongly typed. All the Java basic types are represented.

Note – The implementation of GC read and write barriers are hidden beneath the Ref typed accessors.

---

macro void **CVMID_arrayReadRef**(CVMExecEnv* ee, CVMArrayOfRefICell* arr, CVMUint32 index, CVMObjectICell* item)

macro void **CVMID_arrayWriteRef**(CVMExecEnv* ee, CVMArrayOfRefICell* arr, CVMUint32 index, CVMObjectICell* item)

macro void **CVMID_arrayReadInt**(CVMExecEnv* ee, CVMArrayOfIntICell* arr, CVMUint32 index, CVMJavaInt item)

```
macro void CVMID_arrayWriteInt(  CVMExecEnv* ee, CVMArrayOfIntICell*
arr,   CVMUint32 index, CVMJavaInt item)


macro void CVMID_arrayReadByte(   CVMExecEnv* ee,
CVMArrayOfByteICell* arr,  CVMUint32 index, CVMJavaByte item)

macro void CVMID_arrayWriteByte(  CVMExecEnv* ee,
CVMArrayOfByteICell* arr,  CVMUint32 index, CVMJavaByte item)


macro void CVMID_arrayReadBool(   CVMExecEnv* ee,
CVMArrayOfBoolICell* arr,  CVMUint32 index, CVMJavaBool item)

macro void CVMID_arrayWriteBool(  CVMExecEnv* ee,
CVMArrayOfBoolICell* arr,  CVMUint32 index, CVMJavaBool item)


macro void CVMID_arrayReadShort(  CVMExecEnv* ee,
CVMArrayOfShortICell* arr, CVMUint32 index, CVMJavaShort item)

macro void CVMID_arrayWriteShort( CVMExecEnv* ee,
CVMArrayOfShortICell* arr, CVMUint32 index, CVMJavaShort item)


macro void CVMID_arrayReadChar(   CVMExecEnv* ee,
CVMArrayOfCharICell*  arr, CVMUint32 index, CVMJavaChar item)

macro void CVMID_arrayWriteChar(  CVMExecEnv* ee,
CVMArrayOfCharICell*  arr, CVMUint32 index, CVMJavaChar item)


macro void CVMID_arrayReadFloat(  CVMExecEnv* ee,
CVMArrayOfFloatICell* arr, CVMUint32 index, CVMJavaFloat item)

macro void CVMID_arrayWriteFloat( CVMExecEnv* ee,
CVMArrayOfFloatICell* arr, CVMUint32 index, CVMJavaFloat item)
```

8.5.2   Accessing Elements of 64-bit Width

These macros use the following parameters:

- the first parameter is a pointer to the execution environment (CVMExecEnv) of the current thread
- the second parameter is a registered indirect array reference (CVMArrayOf<T>ICell*)
- the third parameter is the array index *where the first array element is at index 0*
- the fourth parameter is an l-value of type CVMJavaVal64 to read into or a value to write

8.5.2.1   Weakly-Typed Versions

The weakly-typed versions read from and write to a word-aligned two-word area pointed to by location:

macro void **CVMID_arrayRead64**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 off, CVMJavaVal32* location)

macro void **CVMID_arrayWrite64**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 off, CVMJavaVal32* location)

where <CVMArrayOf64ICell> is either CVMArrayOfLongICell or CVMArrayOfDoubleICell.

8.5.2.2   Strongly-Typed Versions

macro void **CVMID_arrayReadLong**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMID_arrayWriteLong**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMID_arrayReadDouble**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 index, CVMJavaVal64 val64)

macro void **CVMID_arrayWriteDouble**(CVMExecEnv* ee, <CVMArrayOf64ICell>* o, CVMUint32 index, CVMJavaVal64 val64)

where <CVMArrayOf64ICell> is either CVMArrayOfLongICell or CVMArrayOfDoubleICell.

8.5.3   Miscellaneous Array Operations

All generic object operations apply to arrays as well. In particular, the header of an array object starts out with an object header, with an additional length entry, so any operation on the header of an object may be performed on an array header.

Below is an array-specific operation.

macro void **CVMID_arrayGetLength**(<CVMArrayOfAnyICell>* o, CVMJavaInt32 len)

where <CVMArrayOfAnyICell>* is a registered indirect array reference, and len is an l-value of type CVMJavaInt32 to store the length result in.

8.5.4 GC-unsafe Operations

The remaining two operations allow setting and getting the referent of an ICell.

Caution – The use of the ICell referent operations is *GC-unsafe*. Therefore, these operations should be used with extreme care in a few selected places, and only within GC-unsafe regions.

Get the referent of `icellPtr`

macro CVMObject* **CVMID_icellDirect**(CVMObjectICell* icellPtr)

Set the referent of `icellPtr` to be the direct object reference `directObj`

macro void **CVMID_icellSetDirect**(CVMObjectICell* icellPtr, CVMObject* directObj)

How to be GC-Safe

This chapter describes how to safely address garbage collection issues within the runtime implementation. In practice, these issues mostly affect implementation issues in the shared source code and not in the porting layer. The techniques in this chapter can be used in the rare cases where it is necessary to be GC-safe in a porting layer.

This chapter covers the following topics:

- Introduction
- Living with ICells
- Explicitly Registered Roots
- GC-safety of Threads

9.1 Introduction

In CDC HI, GC may be running at any time during program execution, searching for program state or changing object addresses. Therefore, you should be very careful whenever you want to access and change Java objects from native code.

The *indirect memory interface* allows you to safely manipulate objects from native code. The calls that make up the indirect memory interface operate on pointers to *ICells* (indirection cells), which are non-moving locations in memory holding object references. ICells must be *registered* with the garbage collector (GC), so they can be found and updated when a GC occurs. Registered ICells may be *local roots*, or *global roots*.

The implementation of the indirect memory interface makes use of a per-thread *GC-safety* flag. Each indirect memory call on an ICell marks the caller thread as *GC-unsafe*, manipulates the Java object encapsulated by the ICell, and marks the thread as *GC-safe* again. Threads that are marked GC-unsafe cannot tolerate GC until they

are marked GC-safe again. GC is only allowed to proceed if all threads are GC-safe. Use of the indirect interface in conjunction with registered ICells makes your C code safe from garbage collection, and makes the GC aware of your Java object use.

Please refer to the chapters Section , "Direct Memory Interface Reference" on page 7-1 and Section , "Indirect Memory Interface Reference" on page 8-1 for reference.

9.2 Living with ICells

ICells and the indirect memory interface form the foundation of the *exactness* architecture of CDC HI. Therefore, it is critical to understand the various ways these calls can be used to ensure GC-safety.

Working on an exact system is different than working on a conservative system. In a conservative system, the GC scans the native stacks and native registers, searching for values that *look like* pointers. So in order to keep a heap object *alive*, it is sufficient to keep around references to it in registers or stack locations for the GC to find them.

In an exact system, all locations holding pointers to heap objects must be known to the GC. There are two types of such known locations:

1. **Implicitly registered locations**: The GC has a *default root scan*, and certain well-known VM data structures are considered to be default roots. Among this group are class statics, string intern tables, JNI tables holding global references, and others. All the references in these data structures are considered *implicitly registered* with GC.

2. **Explicitly registered locations**: Any ICells that are not included in the default root scan of GC must be explicitly registered, either by the local roots mechanism or the global roots mechanism.

9.2.1 ICell Types

ICells encapsulate direct heap object references. Heap objects may be regular Java objects or arrays. There are different ICell types to express each. ICells for all non-array object types are declared as `CVMObjectICell`. Arrays of different Java types have corresponding ICell types. For basic type `<T>`, the right ICell type is `CVMArrayOf<T>ICell`. Other ICell types are:

```
CVMObjectICell        ocell;   /* for CVMObject references        */


CVMArrayOfByteICell   acellb;  /* for CVMArrayOfByte references   */
```

```
CVMArrayOfShortICell    acells;  /* for CVMArrayOfShort references  */
CVMArrayOfCharICell     acellc;  /* for CVMArrayOfChar references   */
CVMArrayOfBooleanICell acellz;  /* for CVMArrayOfBoolean references*/
CVMArrayOfIntICell      acelli;  /* for CVMArrayOfInt references    */
CVMArrayOfRefICell      acellr;  /* for CVMArrayOfRef references    */
CVMArrayOfFloatICell    acellf;  /* for CVMArrayOfFloat references  */
CVMArrayOfLongICell     acelll;  /* for CVMArrayOfLong references   */
CVMArrayOfDoubleICell  acelld;  /* for CVMArrayOfDouble references */
```

Because ICells contain references that may be manipulated by GC, their referents should be set, nulled, and assigned to one another using calls from the indirect memory interface (see Section 8.2, "ICell Manipulations" on page 8-1. Their values should only be passed around as ICell* to ensure GC-safety. So given ocell1 and ocell2 of type CVMObjectICell:

```
CVMObjectICell* ocell1;

CVMObjectICell* ocell2;

CVMExecEnv*     ee = CVMgetEE();

CVMBool res;


<... make sure ocell1 and ocell2 point to registered ICells. They
could be local roots or global roots, for example. See below ...>
CVMID_icellSetNull(ee, ocell1);
CVMID_icellSetNull(ee, ocell2);


CVMassignDirectReferenceTo(ee, ocell1);
CVMID_icellIsNull(ee, ocell1, res);
if (!res) {
   /* Assign the referent of ocell1 to the referent of ocell2 */
   CVMID_icellAssign(ee, ocell2, ocell1);
}
```

In the example above, the only values passed around are *pointers* to ICells. Any assignment to the encapsulated direct object reference of an ICell (as assignDirectReferenceTo() does) must happen in a *GC-unsafe* region, created in the body of the implementation of CVMID_icellAssign(). GC-unsafe regions are explained in the section, Section 9.4, "GC-safety of Threads" on page 9-11 .

9.3 Explicitly Registered Roots

Heap object references that are not part of the default root scan of garbage collection need to be explicitly registered with the collector. There are two separate mechanisms for explicit registration:

- **Local roots:** These are short-lived values, like local variables. They are allocated and deallocated in a stack-like fashion. The interface for using them is geared towards fast allocation and deallocation, and does not allow out-of-order deallocation.
- **Global roots:** These are long-lived values, like global variables. The program can obtain registered global root locations through the global roots API. Global roots may be created and destroyed out-of-order.

9.3.1 Declaring and Using Local Roots

Local roots are an efficient way of declaring, registering and unregistering ICells of local scope. They are typically used to hold relatively short-lived values; think of them as GC-registered local variables. Also note that local roots are thread-local; they are created, used and discarded in the same thread.

The use pattern is the following:

```
//
// Start a local root block, passing in the current 'ee'
// (execution environment), which contains per-thread
// information.
//
CVMID_localrootBegin(ee); {
    CVMID_localrootDeclare(Type1ICell, var1);
    CVMID_localrootDeclare(Type2ICell, var2);
    //
    // use var1 and var2 as Type1ICell* and Type2ICell*
    // respectively
    //
    // do NOT leave the block without executing
    // CVMID_localrootEnd()!
    //
} CVMID_localrootEnd();
```

Since local roots occur more often (dynamically) than global roots, the interface for using local roots is optimized for allowing stack-like fast allocation and deallocation. Conceptually:

1. `CVMID_localrootBegin()` marks the beginning of a scope containing a list of `CVMID_localrootDeclare()` calls.

2. The implementation keeps track of `CVMID_localrootDeclare()` calls. For each, it allocates and registers a local ICell and declares an `ICell` pointer to that registered ICell.

3. When the programmer is done with the local roots in the scope, he/she calls `CVMID_localrootEnd()`, which discards all allocated local roots in that scope.

Note that it is important to call `CVMID_localrootEnd()` when leaving a local root scope; this call discards all registered local roots declared since the last `CVMID_localrootBegin()`. Also note that `CVMID_localrootBegin()` and `CVMID_localrootEnd()` may nest arbitrarily.

9.3.1.1   Example of Local Root Use

This example illustrates local root use. You want to call an allocating operation that is possibly a few functions deep. Therefore you want the caller to declare a local root, and pass its corresponding `ICell*` as a result argument to the operation. This keeps the allocated object safe from garbage collection the moment it is stored in the result argument. When the operation is complete, the caller can unregister the local root.

The following creates a Java string from a Utf8 string. It is an inlined (fast) version of the String constructor. It uses two local roots for temporary values, and discards them after a String has been successfully created and assigned to a `result` ICell.

```
void CVMmakeStringFromUtf8(CVMUtf8* chars, CVMObjectICell* result) {

   CVMID_localrootBegin(); {

      // Two local roots to be used as temporaries
      CVMID_localRootDeclare(CVMObjectICell,      string);
      CVMID_localRootDeclare(CVMArrayOfCharICell, theChars);


      CVMJavaInt length;


      // Make the string object
      CVMID_objectNewInstance (CVMjavaLangStringClassblock, string);


      // .. . and the chars array
```

```
       // Pass the local root in to receive the resulting char[]
       CVMmkArrayOfCharFromUtf8 (chars, theChars);

       CVMID_arrayGetLength (theChars, length);

       //
       // Assign the values of the string
       //
       CVMID_fieldWriteRef(string,
       CVM_offsetOf_java_lang_String_value, theChars);

       CVMID_fieldWriteInt(string,
       CVM_offsetOf_java_lang_String_length, length);

       CVMID_fieldWriteInt(string,
       CVM_offsetOf_java_lang_String_offset, 0);

       // We write the result back to the result ICell.
       CVMID_icellAssign  (result, string);

       // We can now discard the local roots, assuming 'result' was
       // a pointer to a registered ICell.
    } CVMID_localrootEnd();
```

A possible caller of this may be the constant resolution code, resolving a constant pool entry of type CONSTANT_String. The result ICell may be the actual constant pool slot, which is updatable by the GC when it scans class information. (In other words, the constant pool slot for a String constant is an implicitly registered ICell).

So the call would be something like:

```
void CVMresolveStringConstant(CVMConstantPool* cp, CVMJavaShort
strIdx, CVMJavaShort utf8Idx)

{
   CVM_CLASS_RESOLUTION_LOCK();
       CVMID_icellSetNull(&cp.entries[strIdx].str);
       //
       // Mark it as being resolved. This way, no thread can
       // yet use this c.p. entry; however GC can scan it
       // if necessary.
```

```
        //
        CVMcpSetBeingResolved(cp, strIdx);
        CVMmakeStringFromUtf8(cp.entries[utf8Idx],
        &cp.entries[strIdx].str);
        CVMcpSetResolved(cp, strIdx);
    CVM_CLASS_RESOLUTION_UNLOCK();
}
```

9.3.2 Declaring and Using Global Roots

Global root registration allows for declaring, registering and unregistering ICells of
global scope. Global roots are typically used to hold long-lived values that are to be
included in the GC root scan; think of them as GC-registered global variables.

The use pattern is the following:

```
//
// Part of CVMglobals
//
struct CVMGlobalState {
    ....
    CVMObjectICell*     globalRoot1;
    CVMObjectICell*     globalRoot2;
    ....
}
...
void CVMinitThisModule()
{
    CVMglobals.globalRoot1 = CVMID_getGlobalRoot();
    CVMglobals.globalRoot2 = CVMID_getGlobalRoot();
    ...
}
...
void CVMuseThisModule()
{
    // globalRoot1 and globalRoot2 may safely be used as
    // ICell* arguments to CVMID_ operations.
```

```
    CVMID_objectNewInstance(CVMclassJavaLangStringClassblock,
CVMglobals.globalRoot2);

    CVMID_icellAssign(CVMglobals.globalRoot1,
CVMglobals.globalRoot2);

}

...

void CVMexitThisModule()

{

    CVMID_freeGlobalRoot(CVMglobals.globalRoot1);

    CVMID_freeGlobalRoot(CVMglobals.globalRoot2);

}
```

Any long-lived ICell declaration should be registered as a global root. These include C structure fields and global variables.

9.3.2.1 Examples of Declaring and Using Global Roots

The following examples show how to register an ICell referred to by a C struct, a C struct that contains a Java pointer, and how to register a well-known value.

Example of Registering an ICell Referred to by a C Struct

This example shows registering an ICell referred to by a C struct. There may be long-lived C structures in the system with heap object references, like a C hash table with a Java array as the list of values. The declaration for that hash table in CVM would be:

```
typedef struct CVMStrIDhash {

    < ... Other hashtable fields ...>

    CVMArrayOfRefICell* params; /* param table, if needed */

} CVMStrIDhash;
```

where the `params` array is declared as an ICell* holding an array of references. We would allocate these `StrIDhash` nodes as follows:

```
/* Create a hash table of the specified size */

static CVMStrIDhash *

CVMcreateHash(int sizeInBytes)

{

    CVMStrIDhash *h;

    h = (StrIDhash *)CVMCcalloc(1, sizeInBytes);
```

```
   if (h != NULL) {
      CVMinitNode(h);
      //
      // Register and null out the value
      //
      h->params = CVMID_getGlobalRoot();
   }
   return h;
}
```

After registration, h->params may be used as a registered ICell* parameter to other CVMID_ operations. So to allocate the params array:

```
CVMBool
CVMmkParams(CVMStrIDhash* hash, int size)
{
   CVMArrayOfRefICell* params = hash->params;
   CVMID_newArrayOfRef(CVMjavaLangObjectClassblock, size, params);
   if (CVMID_icellIsNull(params)) {
      return CVM_FALSE; // Allocation failed
   } else {
      return CVM_TRUE;
   }
}
```

Example of a C Structure that Contains a Java Pointer

This example shows a C structure that contains a Java pointer. This declaration is from the CVMClassblock structure. Whenever a new CVMClassblock is allocated, the ICell* typed fields are initialized to point to fresh global roots:

```
struct CVMClassblock {
   ...
   CVMObjectICell* classLoader;
   ...
};
typedef struct CVMClassblock CVMClassblock;
...
CVMClassblock* class   =(CVMClassblock*)CVMCcalloc(1,
sizeof(CVMClassblock));
```

```
//
// Get a new, nulled global root to hold a classloader
// reference
//
class->classLoader     = CVMID_getGlobalRoot();
//
// Make a new ClassLoader instance, and assign it to its
// location in class 'class'.
//
CVMID_objectNewInstance(CVMglobals.javaLangClassLoaderClassblock,
class->classLoader);
```

When the `CVMClassblock` is freed, all its registered global roots must be freed first:

```
void CVMclassUnload(CVMClassblock* class)
{
   //
   // Free all class-related data structures
   //
   ...
   //
   // Now get rid of the global roots
   //
   CVMID_freeGlobalRoot(class->classLoader);


   // And finally the Classblock itself
   CVMCfree(class);
}
```

Example of Registering a Well-known Value

This example shows how to register well-known values. Let's assume that we want to have global instances of the `java.lang.Class` versions of some Java classes. Note that CDC HI would not necessarily do this, since `Classblocks` are not allocated on the heap, but it is a good example for global roots.

```
//
// Part of CVMglobals
//
struct CVMGlobalState {
```

```
   ....
   CVMObjectICell* classJavaLangObject;
   CVMObjectICell* classJavaLangString;
   ....
}
...
CVMinitVM()

   /* Allocate and null out global roots */
   CVMglobals.classJavaLangObject = CVMID_getGlobalRoot();
   CVMglobals.classJavaLangString = CVMID_getGlobalRoot();

   // ... and they lived happily ever after
   CVMfindSystemClass("java/lang/Object",
CVMglobals.classJavaLangObject);
   CVMfindSystemClass("java/lang/String",
CVMglobals.classJavaLangString);
}
```

9.4   GC-safety of Threads

Each thread in CDC HI has a flag called the *GC-safety flag*. Whenever a thread performs an operation that manipulates heap objects directly, it is marked as GC-unsafe. If another thread initiates a GC at this time, all threads must be rolled to GC-safe points in order for GC to proceed safely.

The bytecode interpreter typically works in a GC-unsafe manner to allow for efficient direct access to heap objects. To bound the time the thread remains GC-unsafe, backwards branches, method calls, and method returns are designated as GC-safe points. At those points each thread polls for GC. If there is a GC request, the thread suspends itself to rendezvous with all the other threads rolling forward to their GC points. Execution continues after GC.

The implementation of the indirect memory interface marks the caller thread GC-unsafe while it is manipulating object references directly. These are typically very simple operations, and result in only a small window of GC-unsafety.

CDC HI also allows arbitrary sets of operations to proceed in GC-unsafe regions. These operations should be bounded in execution time and are not allowed to block.

For the full set of GC-safety operations, see .

9.4.1   GC-atomic Blocks

If you want GC disallowed while executing a certain set of operations, use:

```
CVMD_gcUnsafeExec(ee,
    <... gc-unsafe code ...>
)
```

where `ee` is a pointer to the execution environment (`CVMExecEnv`) of the current thread.

The GC-unsafe code may not block, perform I/O, or otherwise take too long to execute, in order to keep the time the GC is disabled to a minimum.

When writing GC-unsafe code, extreme care must be taken to avoid calls to arbitrary library routines. These may take too long to execute, or grab platform locks that might end up blocking. The example of `malloc()` comes to mind. So make sure you become GC-safe before making such a call (see Section 9.4.2, "Offering a GC-safe Point" on page 9-14).

Direct pointers to objects may be used within the unsafe block; however, you should make sure that all direct values are written back into registered ICells before exiting the `gcunsafe` block.

Example 1:

Let's say that you want to use two direct memory accesses consecutively without the overhead of being GC-unsafe around each. Here's how you would do that:

```
CVMObjectICell* cell1;
CVMObjectICell* cell2;
CVMJavaInt      val1, val2;

...
< Assume cell1 and cell2 point to registered ICells >
...
CVMD_gcUnsafeExec(ee, {
    CVMObject* o1 = CVMID_icellDirect(cell1);
    CVMObject* o2 = CVMID_icellDirect(cell2);
```

```
    // Read the third integer field of
    // each object.
    CVMD_fieldReadInt(o1, 2, val1);
    CVMD_fieldReadInt(o2, 2, val2);
} ); // End of gcunsafe region
```

Example 2:

Setting the referent of an ICell. The example below takes an ICell* for a char[]
array, allocates a string, starts a GC-unsafe region, and calls out to initialize the
string's fields. The ICell pointed to by resultString gets a reference to the
allocated string before the GC-unsafe region is exited.

Note that this sort of long GC-unsafe region is intended as an example only; this
style should only be used in performance critical points, where direct accesses help
make the code faster.

```
void CVMmakeString(CVMArrayOfCharICell* theChars, CVMObjectICell*
resultString) {
    CVMID_localrootBegin(ee); {
        CVMID_localrootDeclare(CVMObjectICell, tempString);
        // Allocate a String
        CVMID_objectNewInstance(CVMjavaLangStringClassblock,
    tempString);

        CVMD_gcUnsafeExec(ee, {
            CVMObject*     str   = CVMID_icellDirect(tempString);
            CVMArrayOfChar* chars = CVMID_icellDirect(theChars);
            CVMJavaInt      a_len;

            CVMD_arrayGetLength(chars, a_len);
            CVMinitializeDirectStringRef(str, chars, a_len, 0);

            // Set the referent of the ICell that resultString points
            to CVMID_icellSetDirect(resultString, str);
        } );
    } CVMID_localrootEnd();
}
```

Example 3:

Performing an operation that modifies a data structure that is a default GC root. In the example below `CVMicellList[]` is a thread-local list of free ICells and is a default GC root. `CVMgetICell()` allocates ICells from the list. All assigned ICells from the list are scanned by the GC during the root scan. The operation needs to disable the GC; otherwise a GC scan might find the ICell list in an inconsistent state.

```
CVMObjectICell CVMicellList[];
CVMUint32       CVMicellListPtr;


CVMObjectICell* CVMgetICell()
{
    CVMObjectICell* ret;
    CVMD_gcUnsafeExec(ee, {
        ret = &CVMicellList[icellListPtr++];
        *((CVMUint32*)ret) = 0;
    } );
    return ret;
}
```

If a GC were to happen after the increment and before the initialization, garbage values might be erroneously scanned by GC, potentially causing a crash. Therefore, to prevent race conditions, `CVMD_gcUnsafeExec()` keeps GC disabled between these two operations.

9.4.2   Offering a GC-safe Point

Code that is GC-unsafe for long segments must periodically *offer* a GC-safe point. For example, the interpreter runs in a GC-unsafe way, manipulating direct pointers, etc., but at backwards branches, at method calls, and maybe other points, it must offer to be GC-safe to bound the time from a GC request to a GC cycle start. Also, long running operations like data structure expansions or lengthy computations must offer to be GC-safe occasionally. And finally, the VM must offer GC-safe points before doing potentially blocking OS calls like dynamic linker resolution for natives, acquiring locks, or I/O in order to make sure there are no blocked, GC-unsafe threads in the system.

The standard pattern of doing this is to use the `CVMD_gcSafeExec()` or the `CVMD_gcSafeCheckPoint()` macros. For details, refer to Section 7.4, "GC-safety of Threads" on page 7-6.

`CVMD_gcSafeCheckPoint()` is used to offer a GC-safe point for operations that
will definitely not block:

```
CVMD_gcSafeCheckPoint(

    ee,

    {

        <Save your state for possible GC>

    },

    {

        <Restore your state after possible GC>

    }

);
```

`CVMD_gcSafeExec()` is used to offer a GC-safe point for operations that might
block.

```
CVMD_gcSafeExec(

    ee,

    {

        <Do potentially blocking operation>

    }

);
```

At the end of one of these macros, the executing thread is once again GC-unsafe.

Example 1:

The interpreter becomes GC-safe on a backwards branch.

```
<...>
case opc_goto: {

    CVMJavaShort skip = CVMgetJavaShort(currentPc + 1);

    if (skip <= 0) {

        CVMD_gcSafeCheckPoint(ee,

            {

                CVM_DECACHE_INTERPRETER_STATE(currentFrame,
                currentPc, currentSp);

            },

            {

                // No reconstruction needed since Java code

                // will not execute in this thread
```

```
                    // between DECACHE_... and GC.
            }
        );
    }
    CVMexecuteNextInstructionAtOffset(skip);
  }
<...>
```

In effect, the `CVMD_gcSafeCheckPoint()` operation polls a global variable to see if a GC is requested. If it is requested, then the state save operation occurs, GC is run, and the state is reconstructed. If no GC was requested, we go on. This is a very efficient way to create polling-based GC-safe points.

Example 2:

Blocking operations need to become GC-safe. Here's a tricky example: a two-part `monitorenter` operation. The first one gets access to the object, and checks to see if blocking is needed. If no blocking needed, there is no need to become GC-safe. If blocking is needed, we save our state, become GC-safe and block.

```
....
case opc_monitorenter: {
    CVMObject* lockedObject;
    vmResult   result;

    lockedObject = STACK_OBJECT(-1);
    CHECK_NULL(lockedObject);
    result = CVMobjectTryLock(lockedObject); // Try to lock
    if (result == VM_LOCKED_WITHOUT_BLOCKING) {
        //
        // The uncontended case
        // We have already succeeded locking
        //
     } else {
        //
        // May now block.
        // Save interpreter state, stash away locked object
        // as a GC-root, and use monitorEnterMayblock() with
        // the ICell version.
```

```
        //
        CVM_DECACHE_INTERPRETER_STATE(currentFrame, currentPc,
currentSpCached);
        CVMD_gcSafeExec(
        ee,
        {
          // Pass in the stack slot as the ICell*
          CVMmonitorEnterMayblock(&STACK_OBJECT(-1));
        }
    )
    }
    if (CVMcheckForException() == VM_NO_POSTED_EXCEPTION) {
        UPDATE_PC_AND_TOS_AND_CONTINUE(1, -1);
    } else {
        CVMhandleException();
    }
}
```

Example 3:

VM becomes GC-safe before making a call that might allocate from the C heap. This routine may end up taking a long time execute, or even worse, block on an OS lock. Therefore the VM needs to become GC-safe before making the call, in a way that allows blocking.

Here's an excerpt from the stack expansion code that becomes GC-safe before attempting to allocate from the C heap.

```
CVMStackVal32*
CVMexpandStack(CVMStack* obj, CVMUint32 capacity)
{
    ....
    * Must allocate new chunk. Take excess capacity into account.
    */
    size = sizeof(CVMStackChunk) + sizeof(CVMStackVal32) * (capacity
- CVM_MIN_STACKCHUNK_SIZE);

    newStackSize = s->stackSize + capacity;
    if (newStackSize > CVM_MAX_STACK_SIZE) {
```

```
            ... throw exception ...
            }
            CVMD_gcSafeExec(ee, {next = (CVMStackChunk*)CVMCmalloc(size);}
            );
            ....
}
```

PART V   GUI Layer

This part describes the porting interfaces for the GUI alternatives.

- Personal Profile
- Personal Basis Profile

Personal Basis Profile

10.1   Introduction

Personal Basis Profile provides a hosting layer for alternate GUI toolkits that are based on Java technology. These GUI toolkits are called *lightweight component toolkits* to distinguish them from *native GUI toolkits* which are based on native platform technology. Lightweight component toolkits can be based on industry-standard or vendor-specific interfaces.

At a lower level, the Personal Basis Profile reference implementation (RI) mentioned in this porting guide has a portability interface that allows it to be modified to support different target platforms. The Personal Basis Profile RI is based on the Qt application development framework (`http://www.trolltech.com`). This makes it a good example of how to structure an implementation as well as a technology demonstration that serves as a reference for TCK verification. Product implementations may be based on modifying the Qt implementation or replacing it with an alternate native GUI toolkit.

In many ways, the functionality and porting interface of Personal Basis Profile is a subset of Personal Profile. See Section 11.1, "Introduction" on page 11-1 for an overview of AWT.

This chapter provides guidance for porting Personal Basis Profile to an alternate native GUI toolkit. See the *Personal Basis Profile TCK Users Guide* for information about how to use the Personal Profile TCK test suite.

10.2   Porting Architecture

The goals of the porting layers of Personal Basis Profile are to:

- Provide native functionality that supports the GUI semantics and imaging model in Personal Basis Profile.
- Meet product requirements for speed and cost/resource constraints.
- In some cases, provide GUI interoperability that allows AWT-based applications to cooperate with native applications.

FIGURE 10-1 describes the basic AWT architecture in Personal Basis Profile. `java.awt` represents the publicly accessible classes that can be used to construct lightweight component toolkits and applications. `sun.awt` contains high-level portion of the Sun AWT implementation. `sun.awt.qt` contains the toolkit support library which implements the native methods of the Qt native bridge.

FIGURE 10-1 Personal Basis Profile AWT Architecture
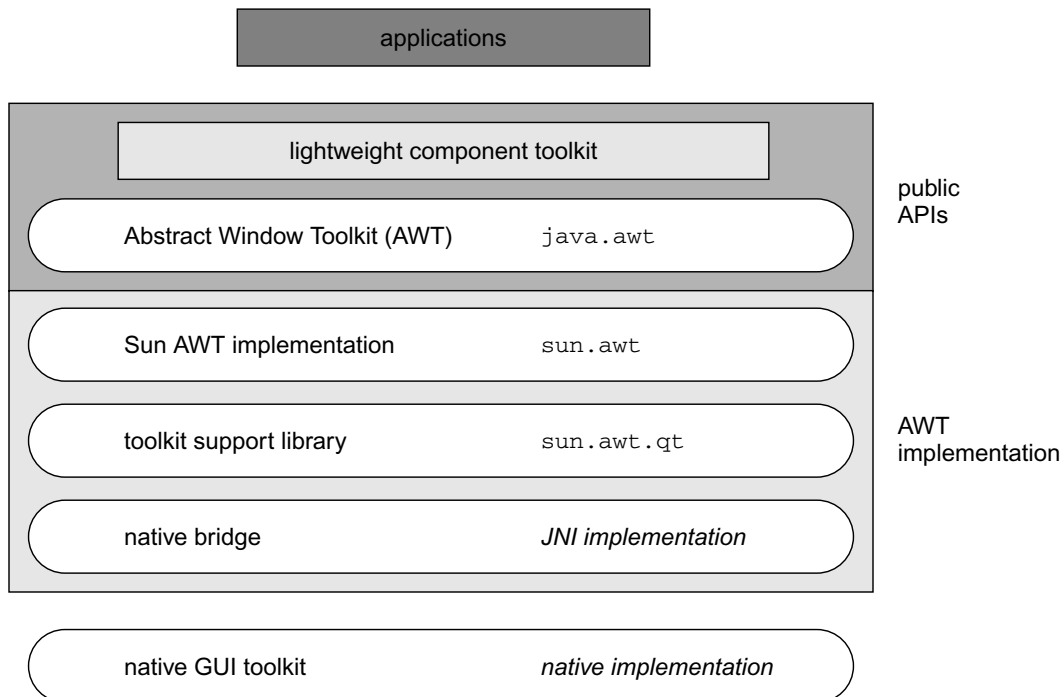
TABLE 10-1 describes the contents of the source code directories that correspond to the layers shown in FIGURE 10-1.

TABLE 10-1   Porting Architecture

Layer	Source Code Directory	Description
public AWT classes	`basis/classes/common/java/awt` `basis/classes/common/java/awt/image`	These public classes are available to lightweight component toolkits and applications.
Sun AWT implementation	`basis/classes/common/sun/awt`	Contains high-level AWT implementation classes. The contents of this directory will not need modification for most ports.
toolkit support library (Java)	`basis/classes/awt/qt/java/awt`	Contains concrete implementations of `QtToolkit`, `QtImage` and other AWT classes. Other than refactoring, the contents of this directory will not need modification for most ports.
native bridge (JNI)	`basis/native/awt/qt`	Contains JNI wrapper functions that call the native GUI toolkit functions. Most of the work during a port involves rewriting the implementations of these JNI wrapper functions.
image decoders (Java)	`basis/classes/common/sun/awt/image`	Contains image handling classes.
native image decoders (JNI)	`basis/native/image`	Contains JNI wrapper functions for `sun.awt.image.`

The files in `basis/native/awt/qt` contain the native bridge source files. This is the central focus of the porting effort.

TABLE 10-2   Native Bridge Source Files (JNI)

File	Description
`QtToolkit.cpp`	JNI wrapper functions for `sun.awt.qt.QtToolkit`, the concrete implementation of `java.awt.Toolkit`.
`Window.cpp`	JNI wrapper functions for the `sun.awt.qt.Window` implementation class.
`QtGraphics.cpp` `QtGraphicsConfiguration.cpp` `QtGraphicsDevice.cpp` `QtGraphicsEnv.cpp`	JNI wrapper functions for the `Graphics` implementation classes.
`QtFontMetrics.cpp`	JNI wrapper functions for the `Font` implementation classes.
`QtImage.cpp`	JNI wrapper functions for the `Image` implementation classes.
`QtSync.cpp` `QtSync.h`	Mutex support.
`QtRobotHelper.cpp`	JNI wrapper functions for `sun.awt.RobotHelper`. NOTE: this is not part of the public API of Personal Basis Profile.
`KeyCodes.h`	Virtual keycode table.
`awt.h`	JNI ID table.

Other useful source files in this directory are `QtSync.cpp` and `QtSync.h` which include Qt-level thread support used in the RI. The OI uses VM-level primitives for thread support.

10.3 Planning and Requirements

10.3.1 Native GUI Toolkit Requirements

An AWT native bridge for Personal Basis Profile can be based on an alternate native GUI toolkit. An advantage of doing this may be the hardware acceleration provided by a native GUI toolkit. A disadvantage may be that the native GUI tookit doesn't include everything necessary to support the AWT imaging model in Personal Basis Profile.

In selecting a native GUI toolkit, the following factors should be considered:

- *Graphics model.* The native GUI toolkit should include a stencil/paint imaging model similar to the one used in `java.awt.Graphics` as well as a device abstraction layer similar to what is found in `java.awt.GraphicsDevice`, `java.awt.GraphicsEnvironment` and `java.awt.GraphicsConfiguration`. Color model support, alpha blending and support for `BufferedImage` will make porting much easier.
- *Truetype font support.*
- *Thread safety.* Thread safety is an important part of any AWT implementation and this is an area where native GUI toolkits can vary quite a bit. On one end of the spectrum are sophisticated native GUI toolkits that have built-in thread management. On the other end are simple native GUI toolkits without internal thread management. The specific properties of your native GUI toolkit will have important implications for the design of a native bridge.
- *Object-oriented design.* Both AWT and the Personal Profile portability layer are object-oriented. Using a native GUI toolkit that has an API with a similar object-oriented design will simplify the porting effort.

At a lower-level, the feature set of a native GUI toolkit has important implications for the design of an alternate implementation of Personal Profile. The following features are important to consider in planning the design of an alternate implementation:

- *Container layout management.* The AWT layout managers like `FlowLayout` can either use a Java implementation or native layout functionality from the native GUI toolkit.
- *Repaint/refresh event management.* Repaint events for some components can be handled by either the native GUI toolkit or with a Java implementation.
- *Event callback registration.* Some native GUI toolkits have functionality for notifying an application when an event happens.
- *Event filtering.* Some native GUI toolkits have functionality for watching the event queue for specific events.

- *Event loop management.* Some native GUI toolkits have their own event loops. Others have only widgets and graphics routines and require an application to supply its own event management.

10.3.2   Notes

- For developers with porting experience from legacy technologies like PersonalJava Application Environment, it is better to start fresh with the Personal Basis Profile implementation rather than attempt to bring over and modify old code. The porting layers described here do *not* use the AWT peer model.
- Thread support is an important part of any AWT implementation because all graphics operations must occur atomically. Using a thread-safe GUI toolkit will simplify the task of supporting this requirement.
- The specification for the `BufferedImage` class in the Personal Basis Profile has been significantly reduced from the J2SE 1.4.2 specification. The reason is that the Java SE version of `BufferedImage` requires too many resources for an embedded application. The Personal Basis Profile version includes only the `getRGB()` and `setRGB()` methods and cannot be directly instantiated or subclassed. Use `GraphicsConfiguration.createCompatibleImage()` to create an instance of `BufferedImage`. If you need to implement `BufferedImage` separately, take a close look at the implementation supplied in the Personal Basis Profile source release to see how it works.

10.4   Writing a Personal Basis Profile AWT Implementation

The following procedures represent the major tasks to perform during the initial port of the Personal Basis Profile source release to an alternate native GUI toolkit.

- Section 10.4.1, "Build System Procedures" on page 10-7 shows how to clone and modify the build system to support an alternate native GUI toolkit.

- Section 10.4.2, "Source Code Procedures" on page 10-7 shows how to clone and modify the Personal Basis Profile source hierarchy to support an alternate native GUI toolkit.

- Section 10.4.3, "Native Bridge Implementation" on page 10-8 outlines some of the tasks in implementing a native bridge.

10.4.1   Build System Procedures

The steps described below are based on cloning the `qt` implementation and creating an implementation called MyGT, for MyGraphicsToolkit. In `build/share`:

1. **Clone the native GUI toolkit-specific build files:**

   ```
   % cp defs_basis_qt.mk defs_basis_mygt.mk
   % cp rules_basis_qt.mk rules_basis_mygt.mk
   ```

2. **Refactor the native GUI toolkit-specific build files to use the** MyGT **prefix.**

10.4.2   Source Code Procedures

1. **Refactor and stub out the JNI wrapper functions in** `basis/native/awt/mygt`**.**

2. **Clone the source file hierarchy.**

   a. **Clone the source files in** `classes/awt/qt`**:**

   ```
   % cp -r qt mygt
   ```

   b. **Clone the source files in** `basis/native/awt/qt`**:**

   ```
   % cp -r qt mygt
   ```

3. **Refactor the source files in** `basis/classes/awt/mygt` **to use the** MyGT **prefix.**

   Since the toolkit support library is fairly high-level, it won't require much modification during a port other than refactoring to use the MyGT prefix. The classes in the toolkit support library mostly handle the task of registering and calling the JNI functions in the native bridge. Therefore the modifications to this layer are pretty straightforward. If you add new source files or change the name of existing source files, be sure to modify the `CLASSLIB_CLASSES` macro in `build/share/defs_basis_mygt.mk` to reflect the changes.

4. **Edit the concrete** `Toolkit` **class.**

   Edit the source file for the concrete `Toolkit` class (e.g. `basis/classes/awt/qt/java/awt/QtToolkit.java`), and change the line that registers the native bridge library to specify the new implementation:

   ```
   new sun.security.action.LoadLibraryAction("mygtawt"));
   ```

5. **Refactor and stub out the JNI wrapper functions in** `basis/native/awt/mygt` **to use the** MyGT **prefix.**

6. **Build the system.**

   After making these changes, the system should build correctly but not run.

   ```
   % make J2ME_CLASSLIB=basis
   ```

10.4.3   Native Bridge Implementation

The JNI source files for the native bridge call library functions in the native GUI toolkit. The first changes to this layer are straightforward and mainly involve refactoring the JNI function implementations and removing unnecessary utility and helper functions. Of course, the main task of porting the Personal Basis Profile source release involves reimplementing these JNI functions with library functions from your native GUI toolkit.

Here are some guidelines for how to approach implementing the native bridge.

- *Event handling code.* You need to start with a solid understanding of the AWT event hierarchy and an appreciation of the tradeoffs between what can be done by the native bridge and the platform toolkit support library. See Section 11.4.4, "Event Delivery Notes" on page 11-11.
- *Thread management code.* The Qt implementation has two cooperating threads: a Java thread created in `sun.awt.qt.QtToolkit` and a Qt-managed native thread created in the `Java_sun_awt_qt_QtToolkit_initIDs` function in `QtToolkit.c`.
- *Graphics code in `QtGraphics*.cc`.* The following JNI methods should be implemented first: `clearRect`, `copyArea`, `drawArc`, `drawLine`, `drawPolygon`, `drawRect`, `drawRoundRect`, `drawString`, `fillArc`, `fillRoundRect`, `removeClip`, `setClip`, `setForeground`, `setPaintMode` and `setXORMode`.

If you add new source files or change the name of existing source files, modify the `AWT_LIB_OBJS` macro in `build/share/defs_basis_mygt.mk` to reflect the changes. You will probably also need to modify the JNI ID table in `awt.h` to reflect any name changes.

Personal Profile

11.1 Introduction

Personal Profile provides a full set of heavyweight AWT components as well as applet support for embedded Web browsers. This provides GUI API compatibility with Java SE applications as well as compatibility with applet-based Web content..

The Personal Profile implementation mentioned in this porting guide is based on the Qt application development framework (http://www.trolltech.com). This makes it a good example of how to structure an implementation as well as a technology demonstration that serves as a reference for TCK verification. Product implementations may be based on modifying the Qt implementation or replacing it with an alternate native GUI toolkit.

This chapter provides guidance for porting Personal Profile to an alternate native GUI toolkit. See the *Personal Profile TCK Users Guide* for information about how to use the Personal Profile TCK test suite.

11.1.1 AWT Background

The Abstract Window Toolkit (AWT) is a basic toolkit for building graphical user-interfaces (GUIs) for Java application software. The AWT divides the task of supplying GUI services between high-level API classes and low-level implementation classes. Java application software interacts only with the high-level API classes while the AWT manages the task of mapping these high-level API classes to implementation classes that supply GUI services through some native GUI toolkit. This architecture allows Java application software to use a single GUI API and run without modification on different target platforms.

GUI widgets like `Buttons` and `Dialogs` can be supplied in two different ways:

- *Lightweight GUI components* are subclassed from `java.awt.Component` and based entirely on Java technology. This allows maximum portability and control by the developer.
- *Heavyweight GUI components* are based on widgets found in native GUI toolkits. This promotes maximum compatibility with native platform technology.

The AWT has been a part of the Java class library from the beginning and since then its public APIs have changed very little. Its role has evolved during the following stages:

- **JDK 1.0.x & 1.1.x.** Originally, the AWT was implemented as a set of high-level API classes and a set of low-level implementation classes. This provided the application portability needed by the Java platform even if it didn't provide a flexible portability strategy for the AWT itself. The main problem was the fact that the AWT was required to supply implementation classes that used a set of public Java interfaces in the `java.awt.peer` package. These peer interfaces simplified the task of mapping API classes to implementation classes but they also limited the flexibility of a given AWT implementation.
- **Swing.** The Swing GUI toolkit provides an alternate GUI framework with a substantially richer interface than the AWT. Most of Swing is implemented as lightweight GUI components that give application developers tremendous flexibility, including the selection of pluggable look & feel designs.
- **PersonalJava.** The PersonalJava Application Environment (PJAE) was based on the early JDK 1.1.x AWT implementation. From an application-level perspective it included changes like optional and modified classes that allow an implementation to require fewer system-level resources. In addition, the PJAE included an AWT implementation that was written almost entirely in Java and did not require a native GUI toolkit or graphics/window system. This changed the portability focus to the device level which was appropriate for the consumer technology market targeted by PJAE.
- **Personal Basis Profile.** The AWT implementation in Personal Basis Profile serves a different purpose than previous AWT implementations. Where earlier AWT implementations included both a fixed set of heavyweight components as well as the ability to create and host lightweight components, the AWT implementation in Personal Basis Profile is intended only to host lightweight component toolkits based on industry-standard or vendor-specific interfaces.
- **Personal Profile.** The AWT implementation in the Personal Profile provides a conventional AWT implementation that is portable and tuned to the needs of resource-constrained devices. This includes both a standard set of heavyweight components and the ability to host lightweight component toolkits. In addition, Personal Profile includes full applet support so that it can be used with embedded Web browsers.

11.1.2   Additional Reading

- *Personal Profile Specification*, JSR-216 - `http://jcp.org/jsr/detail/216.jsp`. The Personal Profile specification describes an API for a AWT-based GUI class library. While the Personal Profile specification does not describe a specific implementation, familiarity with this API specification is still a necessary part of the porting process.
- *Personal Basis Profile Specification*, JSR-217 - `http://jcp.org/jsr/detail/217.jsp`. The Personal Basis Profile specification describes an API for providing a hosting layer for alternate GUI toolkits that are themselves based on Java technology.
- *Java Native Interface: Programmer's Guide and Specification* by Sheng Liang. Addison-Wesley, 1999. The JNI is the main interface for writing the wrapper functions for a native bridge.
- *The Java Class Libraries, Second Edition, Volume 2*, by Patrick Chan and Rosanna Lee. Addison-Wesley, 1998. This is probably the best reference manual for the AWT because it provides much more detail about the AWT classes than the Java SE or Java ME specifications. Though it is based on JDK 1.1 and thus doesn't describe the newer Java SE graphics classes included in Personal Profile, it does provide a more detailed description of the event and peer hierarchies.

11.2   Porting Architecture

The goals of the porting layers of Personal Profile are to:

- Provide native functionality that supports the GUI semantics and imaging model in Personal Profile.
- Meet product requirements for speed and cost/resource constraints.
- In some cases, provide GUI interoperability that allows AWT-based applications to cooperate with native applications.

FIGURE 11-1 describes the basic AWT architecture in Personal Profile. `java.awt` represents the publicly accessible classes that represent heavy weight GUI components. `sun.awt.peer` contains the AWT peer interfaces that shape the

requirements of a concrete AWT implementation. `sun.awt` contains high-level portion of the Sun AWT implementation. And `sun.awt.qt` contains the toolkit support library which implements the native methods of the Qt native bridge.

FIGURE 11-1 Personal Profile AWT Architecture
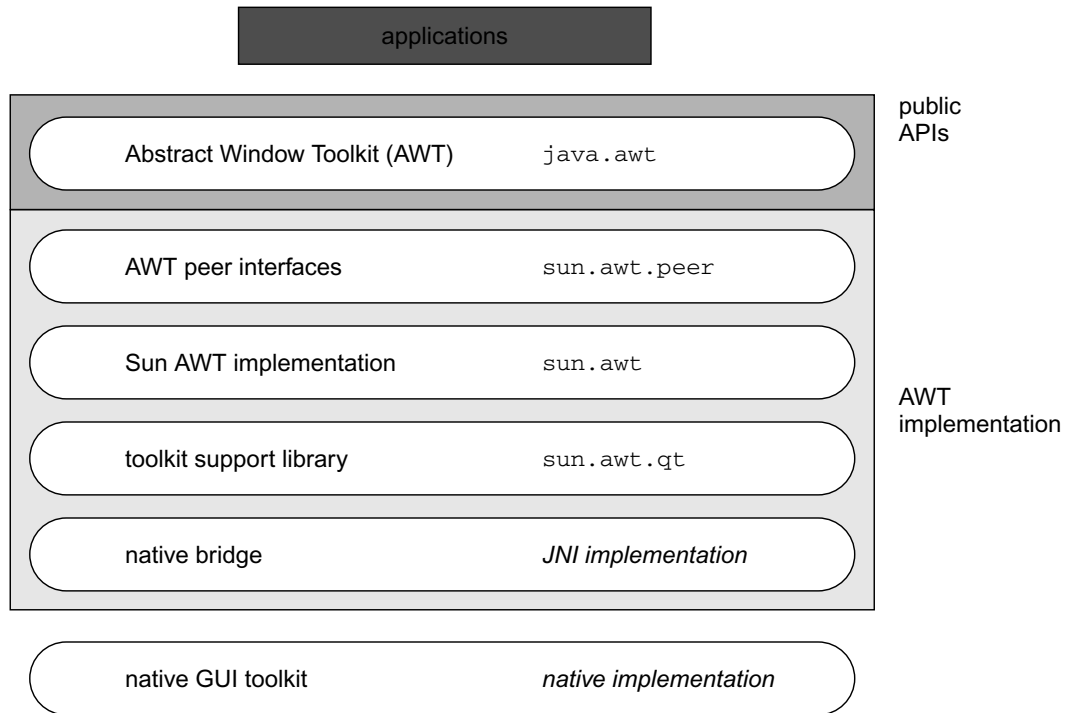
The high-level architecture of a Personal Profile AWT port is organized into the stack of layers described in .

TABLE 11-1   AWT Porting Layers

Layer	Source Code Directory	Description

public AWT classes	`personal/classes/awt/peer_based/java/awt`
	`personal/classes/common/java/awt/`
	`basis/classes/common/java/awt`
		The public AWT classes. Some Personal Profile classes are supersets of the equivalent J2SE 1.4.2 AWT classes. These include some extra functionality to provide backward compatibility for the PersonalJava Application Environment.

peer interfaces (Java)	`personal/classes/awt/peer_based/sun/awt/peer`	The AWT peer interfaces describe the requirements for an AWT implementation. In Personal Profile, these interfaces have been moved from the public `java.awt package` into the implementation-specific `sun.awt.peer` package.
Sun AWT implementation (Java)	`personal/classes/awt/peer_based/sun/awt` `basis/classes/common/sun/awt`	Contains high-level AWT implementation classes. The contents of this directory will not need modification for most ports.
toolkit support library (Java)	`personal/classes/awt/peer_based/sun/awt/qt`	The Qt toolkit support library provides a concrete implementation of `java.awt.Toolkit` which maps AWT component classes to peer classes that encapsulate native widgets that are managed by the Qt-based native bridge.
native bridge (JNI)	`personal/native/awt/qt`	The Qt native bridge is a set of JNI wrapper functions that call Qt library functions.
image decoders (Java)	`personal/classes/common/sun/awt/image`	Contains image handling classes.
native image decoders (JNI)	`personal/native/image`	Contains JNI wrapper functions for `sun.awt.image`.

TABLE 11-2 describes the toolkit support library source files in `personal/classes/awt/peer_based/sun/awt/qt`. The main purpose of the toolkit support library is to supply a concrete implementation of the abstract `java.awt.Toolkit` class and map GUI component classes to peer implementations. The Personal Profile source release includes two other abstract classes, `sun.awt.SunToolkit` and `sun.awt.PeerBasedToolkit`, that simplify this task. For most implementations, it is best to refactor the Qt implementation. The top-level `Toolkit` class also creates a thread for processing events for the toolkit support library.

TABLE 11-2 Qt Toolkit Support Library

Source File	Description
`QtToolkit.java`	The main concrete implementation of `java.awt.Toolkit`.
`Qt*Peer.java` `QtMenuContainer.java`	Implementation classes for peer-based GUI component.
`QtFont.java`	`Font` implementation class.
`Qt*Graphics*.java`	`Graphics` implementation classes.
`Qt*Image*.java`	`Image` implementation classes.
`QtRobotHelper.java`	Implementation class for `sun.awt.RobotHelper`.
`QtClipboard.java`	System-based clipboard support used by `java.awt.datatransfer.Clipboard`.

TABLE 11-3 describes the JNI source files in `personal/native/awt/qt`. This is the main directory for a native bridge implementation and represents the bulk of the work for a porting effort.

TABLE 11-3 Qt Native Bridge

Source File	Description
`awt.cc` `awt.h`	Utility functions and the JNI ID table.
`QtToolkit.cc` `QtToolkit.h` `QtToolkitEventHandler.cc`	JNI wrapper functions for `sun.awt.qt.QtToolkit`, the concrete implementation of `java.awt.Toolkit`.
`Qt*Peer.cc` `Qt*Peer.h`	JNI wrapper functions for the peer-based GUI component classes.
`QtGraphics.cc` `QtGraphics.h`	JNI wrapper functions for the `Graphics` classes.

TABLE 11-3   Qt Native Bridge

Source File	Description

`ImageRepresentation.h`
`QtImageDecoder.cc`
`QtImageRepresentation.cc`
`OffScreenImageSource.cc`
	JNI wrapper functions for the `Image` classes.

`QtClipboard.cc` `QtClipboard.h`	JNI wrapper functions for system-based clipboard support needed by `java.awt.datatransfer.Clipboard`.
`QtKeyboardFocusManager.cc`	JNI wrapper functions for the `java.awt.KeyboardFocusManager`.
`QtRobotHelper.cc`	JNI wrapper functions for `sun.awt.RobotHelper`.
`KeyCodes.cc` `KeyCodes.h`	Table of key combinations generated for `sun.awt.qt.QtTextField` and `sun.awt.qt.QtTextArea`.
`QtSync.cc` `QtSync.h`	Mutex support.

The other source files in `personal/native/qt` contain helper classes that are Qt-specific and therefor not useful during a port.

11.3 Planning and Requirements

11.3.1 Native GUI Toolkit Requirements

In selecting a native GUI toolkit, the following factors should be considered:

■ *Widget set.* The widgets supplied by the native GUI toolkit should match the requirements of the public GUI component classes in the AWT. Most of the GUI components in AWT are straightforward and are in most commonly available native GUI toolkits. The most complex is `FileDialog`. If this or any other component is not available, it may be necessary to implement it separately. Here the choices are between building a lightweight component or perhaps extending the native GUI toolkit.

■ *Graphics model.* The native GUI toolkit should include a stencil/paint imaging model similar to the one used in `java.awt.Graphics` as well as a device abstraction layer similar to what is found in `java.awt.GraphicsDevice`, `java.awt.GraphicsEnvironment` and `java.awt.GraphicsConfiguration`. Color model support, alpha blending and support for `BufferedImage` will make porting much easier.

- *Truetype font support*.
- *Event handling.* The AWT uses a delegated event handling model where an event source creates an event which is then handled by another object. Events are associated with generic user-input devices like `Buttons` and `Scrollbars`. See Section 11.4.4, "Event Delivery Notes" on page 11-11 for information about how native events are scheduled with the AWT event handling mechanism.
- *Thread safety.* Thread safety is an important part of any AWT implementation and this is an area where native GUI toolkits can vary quite a bit. On one end of the spectrum are sophisticated native GUI toolkits that have built-in thread management. On the other end are simple native GUI toolkits without internal thread management. The specific properties of your native GUI toolkit will have important implications for the design of a native bridge.
- *Object-oriented design.* Both AWT and the Personal Profile portability layer are object-oriented. Using a native GUI toolkit that has an API with a similar object-oriented design will simplify the porting effort.

At a lower-level, the feature set of a native GUI toolkit has important implications for the design of an alternate implementation of Personal Profile. The following features are important to consider in planning the design of an alternate implementation:

- *Container layout management.* The AWT layout managers like `FlowLayout` can either use a Java implementation or native layout functionality from the native GUI toolkit.
- *Repaint/refresh event management.* Repaint events for some components can be handled by either the native GUI toolkit or with a Java implementation.
- *Event callback registration.* Some native GUI toolkits have functionality for notifying an application when an event happens.
- *Event filtering.* Some native GUI toolkits have functionality for watching the event queue for specific events.
- *Event loop management.* Some native GUI toolkits have their own event loops. Others have only widgets and graphics routines and require an application to supply its own event management.

11.3.2   Notes

- For developers with porting experience from legacy technologies like PersonalJava Application Environment, it is better to start fresh with the Personal Profile implementation rather than attempt to bring over and modify old code. The porting layers described here are different than peer-based implementations.
- The Personal Profile specification for `java.awt.image.AlphaComposite` allows an implementation to approximate the result of `SRC_OVER`. The implementation in the Personal Profile source release provides a very limited implementation of `SRC_OVER`.
- Personal Profile inherits its API specification for the `BufferedImage` class from Personal Basis Profile. See Section 10.3.2, "Notes" on page 10-6 for more details.

11.4   Writing a Personal Profile AWT Implementation

The following procedures represent the major tasks to perform during the initial port of the Personal Profile source release to an alternate native GUI toolkit. Section 11.4.1, "Build System Procedures" on page 11-9 shows how to modify the build system and clone the source trees for a toolkit support library and native bridge. Section 11.4.2, "Source Code Procedures" on page 11-9 outlines the task of writing a toolkit support library and native bridge based on an alternate native GUI toolkit.

11.4.1   Build System Procedures

The steps described below are based on cloning the `qt` implementation and creating an implementation called MyGT, for MyGraphicsToolkit.

In `build/share`:

1. **Edit the build file** `defs_personal_peer_based.mk` **and change the definition of** `AWT_PEERSET` **to refer to** `mygt`:

   ```
   AWT_PEERSET=mygt
   ```

2. **Clone the build files:**

   ```
   % cp defs_personal_qt.mk defs_personal_mygt.mk
   % cp rules_personal_qt.mk rules_personal_mygt.mk
   ```

3. **Refactor the build files to use the** `MyGT` **prefix.**

11.4.2   Source Code Procedures

The steps described below are based on cloning the `qt` implementation and creating an implementation called MyGT, for MyGraphicsToolkit.

In `src/share`:

1. **Clone the source file hierarchy.**

   a. **Clone the source files in** `classes/awt/peer_based/sun/awt/qt`:

   ```
   % cp -r qt mygt
   ```

   b. **Clone the source files in** `classes/awt/peer_based/sun/awt/qt`:

```
% cp -r qt mygt
```

   c. **Clone the source files in** `native/awt`:

```
% cp -r qt mygt
```

2. **Refactor the source files in** `personal/classes/awt/peer_based/sun/awt/qt`
   to use the `MyGT` **prefix.**

   Since the toolkit support library is fairly high-level, it won't require much
   modification during a port other than refactoring to use the `MyGT` prefix. The classes
   in the toolkit support library mostly handle the task of registering and calling the
   JNI functions in the native bridge. Therefore the modifications to this layer are pretty
   straightforward. If you add new source files or change the name of existing source
   files, be sure to modify the `CLASSLIB_CLASSES` macro in
   `build/share/defs_personal_mygt.mk` to reflect the changes.

3. **Edit the concrete** `Toolkit` **class.**

   Edit the source file for the concrete `Toolkit` class (e.g.
   `classes/awt/peer_based/sun/awt/mygt/MyGTToolkit.java`), and change
   the line that registers the native bridge library to specify the new implementation:

   ```
   new sun.security.action.LoadLibraryAction("mygtawt"));
   ```

4. **Refactor and stub out the JNI wrapper functions in** `personal/native/awt/mygt`
   to use the `MyGT` **prefix.**

5. **Build the system.**

   After making these changes, the system should build correctly but not run.

   ```
   % make J2ME_CLASSLIB=personal AWT_IMPLEMENTATION=peer_based \
       AWT_PEERSET=mygt
   ```

11.4.3   Native Bridge Implementation

The JNI source files for the native bridge call library functions in the native GUI
toolkit. The first changes to this layer are straightforward and mainly involve
refactoring the JNI function implementations and removing unnecessary utility and
helper functions. Of course, the main task of porting the Personal Profile source
release involves reimplementing these JNI functions with library functions from
your native GUI toolkit.

Here are some guidelines for how to approach implementing the native bridge.

- *Peer component/native widget mapping in* `QtToolkit.java` *and* `Qt*Peer.java`. If
  you understand the AWT component hierarchy, this is a straightforward task.
- *Event handling code in* `QtToolkit.c` *and* `Qt*Peer.[ch]`. You need to start with a
  solid understanding of the AWT event hierarchy and an appreciation of the
  tradeoffs between what can be done by the native bridge and the platform toolkit
  support library. See Section 11.4.4, "Event Delivery Notes" on page 11-11.

- *Thread management code.* The Qt implementation has two cooperating threads: a Java thread created in `sun.awt.qt.QtToolkit` and a Qt-managed native thread created in the `Java_sun_awt_qt_QtToolkit_initIDs` function in `QtToolkit.c`. See the description of *Thread Safety* for more information about the tradeoffs and design choices for thread management.
- *Drawing, clipping, cursor and font handling code in `QtGraphics.[ch]`.* The following JNI methods must be implemented: `clearRect`, , `drawArc`, `drawLine`, `drawPolygon`, `drawRect`, `drawRoundRect`, `drawString`, `fillArc`, `fillRoundRect`, `removeClip`, `setClip`, `setForeground`, `setPaintMode` and `setXORMode`.
- *Image handling code in `QtImageRepresentation.c` and `ImageRepresentation.h`.*

If you add new source files or change the name of existing source files, modify the `AWT_LIB_OBJS` macro in `build/share/defs_personal_mygt.mk` to reflect the changes. You will probably also need to modify the JNI ID table in `awt.h` to reflect any name changes.

11.4.4   Event Delivery Notes

The event delivery mechanism for heavyweight components in Personal Profile and other AWT implementations can present some difficulties for a porting engineer. There are two basic problems: the order of event delivery and the mechanism for delivering events from a native widget to the relevant AWT component.

11.4.4.1   Event Ordering

The Personal Profile and Java SE specifications for `java.awt.EventQueue` make it clear that the order of event delivery is implementation-dependent. The notes below describe the implementation approach used in the Personal Profile source release.

- *General Rules*
  - All native mouse and keyboard events must be redirected through the AWT event queue for possible processing and consumption.
  - A series of `MOUSE_MOVED` events may be coalesced.
  - A series of `MOUSE_DRAGGED` events may be coalesced.
  - A series of `PAINT` events will be coalesced.
  - A native widget like a Checkbox could receive an event combination like `MOUSE_PRESS`, `ITEM_EVENT` and `MOUSE_RELEASED`. The order in which these events appear in the Java event queue is not specified.
  - `WINDOW_ICONIFIED` and `WINDOW_DEICONIFIED` events are only generated when the `Frame` is visible.

- *Well-Paired Event Combinations.* The following event combinations should be "well-paired", which means that they should match 1:1 and follow a specific order. For example, for each KEY_PRESS event there should follow a KEY_RELEASED event.
  - MOUSE_ENTER and MOUSE_EXIT
  - WINDOW_ACTIVATED and WINDOW_DEACTIVATED
  - MOUSE_PRESSED and MOUSE_RELEASED
  - KEY_PRESS and KEY_RELEASED
  - KEY_TYPED and KEY_RELEASED
- *Other Rules*
  - A FOCUS_LOST temporary event should be issued for each WINDOW_DEACTIVATED event.
  - A FOCUS_GAINED permanent event should be issued for each WINDOW_ACTIVATED event.
  - A WINDOW_ACTIVATED event should be issued before a FOCUS_GAINED event.
  - A FOCUS_LOST event should be issued before each WINDOW_DEACTIVATED event.
  - A KEY_RELEASED event should be delivered before a KEY_TYPED event.

11.4.4.2   Event Delivery Mechanism

The more difficult problem is in making sure that events are sent to the right component depending on the state of the system. The events that cause the most problems are FOCUS_GAINED, FOCUS_LOST, MOUSE_DRAGGED and MOUSE_MOVED.

- *Focus Events.* FOCUS_GAINED and FOCUS_LOST events are perhaps the hardest to get right but are actually very simple when you fully understand what the peer classes expect. Here are some useful guidelines:
  - KEY_PRESSED, KEY_RELEASED and KEY_TYPED events should be sent to the heavyweight component that has focus. In effect the peers can assume there is one component that always has focus in every containing Window. If no components actually have focus then you can assume the containing Window requested focus and send key events to it.
  - Whenever the focus changes from one component to another, post a FOCUS_LOST permanent event for the component with current focus (see the technique described above for the case where no component has focus) and a FOCUS_GAINED permanent event for the component with new focus. The FOCUS_GAINED permanent event should only be posted if the containing window is currently active. If the containing window is inactive, the FOCUS_GAINED permanent event should not be posted until the containing window becomes activated and after posting the WINDOW_ACTIVATED event.
  - If a window is deactivated then a FOCUS_LOST temporary should be posted to the currently focused component.

- The peer implementation will request focus on a Panel in order to forward events to any lightweight components it contains. This means that heavyweight components that may not be able to request focus from the native widget library should still work. For example, some native GUI toolkits may not allow a `Panel` to get focus and send events to the top level window.
- *Mouse Events.* `MOUSE_MOVED` events are always sent to the heavyweight component under the mouse cursor. `MOUSE_DRAGGED` events are always sent to the heavyweight component where the mouse button was first pressed, even when the mouse is moved outside of the component. The coordinates are always relative to the component that was clicked on.

PART VI   AMS Layer

This part describes the porting interfaces for CDC Application Management System.

■ CDC Application Management System

CDC Application Management System

12.1   Overview

The CDC Application Management System (CDC AMS) is a system for deploying, launching and managing applications, as well as managing system-related tasks like preferences and inter-application communication. This section provides a brief overview of CDC AMS, its architecture and how it can be tuned and modified.

This chapter is organized into the following sections:

- Conventional Java Application Management
- `mtask`: Process-Based JVM Cloning
- CDC AMS Application Management Framework
- Tuning Notes
- Implementation Notes

12.2   Conventional Java Application Management

From the very beginning, Java technology has supported different application models including the standalone `main()`-based application model and various managed application models. Typically, a managed application model allows an application manager that can manage multiple applications. The application

manager and the managed applications can all run in the same Java virtual machine instance. By offloading the tasks of deployment and resource management onto the application manager, managed application models streamline an application's development and simplifies its operation.

FIGURE 12-1 Conventional Application Management

mapp1	mapp2	mapp3
application manager		
Java virtual machine		
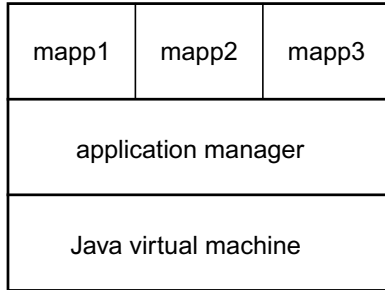
Both the original applet managed application model and the more recent xlet application model use thread management techniques to allow a single Java virtual machine instance to support multiple managed applications. This provides the benefit of scalability but doesn't provide the necessary robustness because a managed application in one thread can potentially crash the entire JVM and any other managed applications with it. Because managed applications are not properly *isolated*, their behavior can affect and be affected by other applications.

One simple way to provide this isolation would be to run each managed application inside in a separate JVM instance and then stitch them together with an application manager in another JVM instance. This approach would definitely provide the

required isolation and robustness, but running multiple JVM instances would have so much resource overhead that it would not achieve the scalability goal and would therefore be impractical.

FIGURE 12-2 Application Management with Separate JVM Instances

mapp1	mapp2	mapp3
JVM	JVM	JVM

AMS
JVM
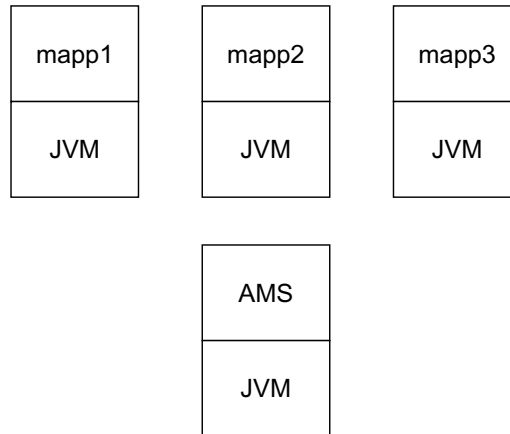
12.3 `mtask`: Process-Based JVM Cloning

The goal of CDC AMS is to provide the isolation, robustness and scalability that is lacking in conventional Java application management systems. The central mechanism is an extension to the JVM called `mtask` which allows a launched JVM in a single OS process to clone itself into a separate operating system process for running an individual managed application. This is functionally equivalent to the process approach described in the previous section. The difference is that `mtask` uses copy-on-write process cloning semantics commonly available in the `fork`(2) system call of the Linux kernel. Copy-on-write shares virtual memory pages between cloned processes until a given page is written to. Then a private copy is made for the cloned process for holding process-specific state. All other shared pages remain unaffected and are still shared.

Here's how `mtask` works. At boot time, CDC AMS launches a JVM instance. This launch procedure performs the normal JVM bootstrapping, heap allocation, data initialization, system class library loading and method compilation. Much of the data that is typically R/W during the launch procedure will quiesce and become R/O. When the JVM launch procedure is complete, the JVM's process memory image contains a fully initialized JVM instance. Finally, the JVM instance puts itself to sleep and waits for messages on a known TCP port.

The term for this sleeping process is the master JVM instance. Its chief purpose is to act as a template for cloning subsequent client JVM instances that will contain the application manager and each of the applications it manages.

After creating a master JVM instance, the first order of business is to create a client JVM instance to contain the application manager. This is done by creating the first clone of the master JVM, and starting the application manager in it. This bootstrap procedure is performed with the aid of a driver utility called cvmc described in the *CDC Runtime Guide*.

The client and server JVM instances now share a great deal of the server JVM instance's memory pages. The major difference is that the client VM instance has consumed some additional memory for the application manager and perhaps some shared pages in the client JVM instance have been overridden with new state. There may be additional changes to JVM data structures, but these are minor. And since the server JVM instance remains unchanged, subsequent client JVM instances that host managed applications will be cloned from this original template and share as much original JVM state as possible.

FIGURE 12-3 Server and Client JVM States
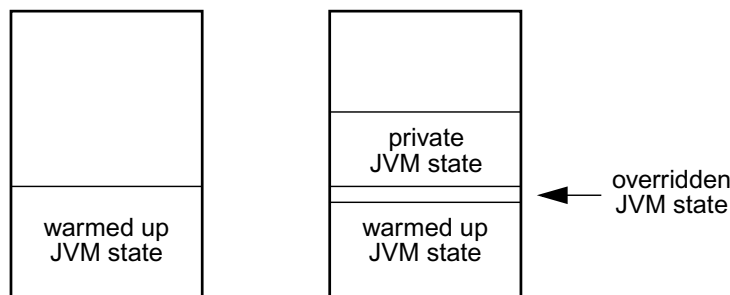
After the application manager becomes operational it takes over the role of sending launch commands to the server JVM instance that had originally been performed by the `cvmc` driver utility. The next step is to create another client JVM instance for hosting a managed application.

FIGURE 12-4 Creating a Client JVM Instance for a Managed Application
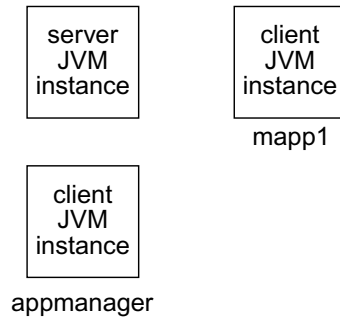
The mechanics are very similar to cloning the client JVM instance for the application manager. The `mtask` JVM extension has a command language for communicating between and controlling client JVM instances. See the *CDC Runtime Guide* for a list of `mtask` commands.

After cloning the client JVM instance, the server JVM instance returns to sleep waiting for a launch command on a known port and the client JVM instance finishes loading the application classes. The managed application now has its own client JVM instance which provides all the process protection that Linux process management provides. Significant memory segments are shared between client JVM instances and the server JVM instance.

It's useful to make a few observations about the client JVM instance and its relationship with the server JVM instance.

- Client JVM instances operate normally. The main difference is that they enter life prewarmed up. This makes each launch cycle much faster than the entire bootstrap process normally associated with a Java virtual machine. However, since the client JVM instance is already warmed up, certain options that are available at launch time are no longer available. For example, the heap is allocated once at startup and cannot be resized in a client JVM instance. Appendix A, cvm Reference describes which command-line options are appropriate for the client JVM instance.

- The main purpose of the server JVM instance is for use as a template for cloning. The server JVM instance does have a small operational component for listening to and acting upon launch commands.

- Resource sharing/shared pages.

- Heap management. The memory pages associated with the heap are cloned using copy-on-write semantics and these pages are shared until they are written to. So long-lived objects are fully shared across multiple JVM instances. A client JVM instance will get a new copy of an object when it is modified.

- Because the heap is cloned with the rest of the JVM instance, its size is fixed at master JVM startup time.

- Reliable termination. Because CDC AMS uses operating system processes to manage applications it is very easy to terminate an application and its JVM instance. If the application has some shared resources that require more graceful handling, like a resource shared with a cooperating application or service, then additional stages of termination will be required.

- `cvmc` is a driver utility for initial cloning and diagnostic purposes only. Subsequent launch commands are handled by the application manager.

- `cvmc` can also be used as a model for integrating CDC AMS with a native AMS.

The CDC AMS architecture is based on an application management infrastructure that uses operating system process management to support concurrent applications as well as an application management framework that provides services for supporting specific application models and deployment mechanisms. The previous sections describe the major components of the process-based application management infrastructure. With this infrastructure it is possible to launch and manage multiple managed applications with isolation, robustness and scalability. This combines the benefits of Linux's well-tested process management with the runtime security of Java technology.

The following sections describe the application management framework and support for a specific application model: xlets.

12.4   CDC AMS Application Management Framework

The application management framework in CDC AMS is a based on a modular, object-oriented design. The goal is not to just support a single application model or user experience model, but instead to provide a framework that can be extended to support different kinds of concrete AMSs for different product designs. The current implementation supports the xlet application model, over-the-air deployment based on J2EE Client Provisioning Specification (JSR-124) and a simple user-experience model. However, the application management framework can be extended to support other application models like xlets or midlets or other deployment mechanisms.

The table below shows the principal modules in the application management framework.

TABLE 12-1   CDC AMS Packages

Package	Description
appmanager	com.sun.appmanager.AppManager is an abstract class that represents a top-level registry for an application manager implementation. This allows different application manager components to communicate with each other through handles maintained in a concrete class that is extended from this abstract class.
appmodel	com.sun.appmanager.appmodel contains abstractions for encapsulating application model lifecycles so they can be represented with mtask. In addition, this package includes a concrete implementation for the xlet application model.
apprepository	The application repository stores packaged applications. The CDC AMS implementation uses a simple hierarchical directory structure in the repository directory of the CDC Java runtime environment.
client	The portion of CDC AMS that is resident in the cloned JVM instance.
mtask	Communication with the mtask server for application launch, message passing, application termination and all other components of the mtask command language.
ota	Supports different OTA protocol implementations. The sample implementation supports the J2EE Client Provisioning Specification (JSR-124).
preferences	Manages persistent user and system preferences. The implementation includes a simple key-value preference mechanism in the repository directory. The default implementation uses backing store in the repository directory.
presentation	The presentation layer manages the user experience and drives the operation of the AMS. The presentation mode accesses the other objects of the AMS through the AppManager registry and can drive execution of the user experience. The application management framework doesn't place a lot of requirements on the presentation layer, just a single interface and a single abstract class. In particular, the user experience is not defined by the application management framework.
store	Defines a simple interface to persistent storage in a device.

`com.sun.appmanager.mtask` and `com.sun.appmanager.client` are central to the architecture of the CDC AMS application manager infrastructure and are not currently intended for extension, modification or porting.

FIGURE 12-5 CDC AMS Architecture
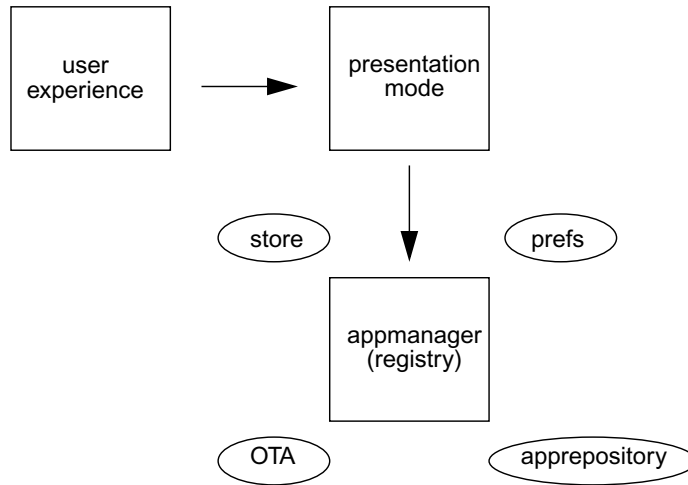
12.5   Tuning Notes

The following sections describe how to create warmup lists for preinitializing classes and precompiling methods.

12.5.1   Generating a Class Preinitialization List

1. **Build a trace-enabled version of the CDC Java runtime environment:**

```
CVM_DEBUG=false
CVM_OPTIMIZED=true
CVM_TRACE=true
```

2. **Run each GUI application with** `-Xtrace:0x00040000`.

This provides a list of classes those are loaded during the execution.

3. **Combine the list from each GUI application together and remove duplicate class entries.**

Try to keep the original class order as much as possible. Don't sort the list.

4. **Identify** `<clinit>`**'s whose effects should not be forked.**

a. **Run the applications with** `clinit` **tracing on (**`-Xtrace:0x2000`**).**

b. **Scan for "**`[InitClass() called for`**" in the trace output.**

This gives you all the classes being initialized, with and without `<clinit>`.

5. **Check if there are classes from the method list not present in the class list.**

Those should be added to the class list.

6. **Test the class preinitialization list.**

Install the class preinitializaton list in the `repository/profile` directory of the CDC runtime environment and then launch CDC AMS with the launch script described in the *CDC Runtime Guide*.

12.5.2   Calculating Code-Cache Size

1. **Build with option** `CVM_JIT_COLLECT_STATS=true`.

2. **Launch CDC AMS with the** `launchams` **script described in the** *CDC Runtime Guide*.

3. **Run an application.**

CDC AMS will print the statistics:

```
Allocated codebuffer bytes       :      458868      96      10868
```

In this example, the code-cache size will be 660k = 460k+ 200k for the application's methods.

12.5.3   Generating a Method Precompilation List

1. **Build a trace-enabled version of the CDC Java runtime environment:**

```
CVM_DEBUG=false
CVM_OPTIMIZED=true
CVM_TRACE=true
```

2. **Run a sample application with the following option:**

   ```
   -Xjit:trace=status,climit=10000
   ```

   `climit` should be half the default to enable aggressive compilation.

3. **Collect the output from** `-Xjit:trace` **for each application.**

   a. **Use** `grep`**(1) to scan for the string** `JS: COMPILED`**.**

   b. **Remove extra words, leaving a pure method signature.**

   c. **Remove application classes and leave library methods.**

   d. **Take the union of all the files and remove duplicates.**

   ```
   % cat file1 file2 file3 | sort -u > methodList.txt
   ```

   e. **Remove any redundant entries.**

   Reduntant entries are defined as a method listed multiple times with differeing return types. Only one version of the particular method is needed because a differing argument list makes a new method, but a differing return type does not.

   Here's an example:

   ```
   java.io.BufferedReader.readLine()
   java.io.BufferedReader.readLine()Ljava/lang/String;
   java.io.BufferedReader.readLine(Z)
   java.io.BufferedReader.readLine(Z)Ljava/lang/String;
   ```

   The first and third lines are appropriate but the second and fourth are redundant.

   The following command line can remove redundant methods:

   ```
   % cat oldmethods.txt | sed 's/).*/)/' | uniq > newmethods.txt
   ```

   Comment lines start with '#', like a shell script.

4. **Test the new method precompilation list.**

   Install the method precompilation list in the `repository/profile` directory of the CDC runtime environment and launch CDC AMS with the launch script described in the *CDC Runtime Guide*.

5. **Check if the new method list includes a class that's not in classlist.**

   The missing classes may need to be added to the classlist for the efficiency.

   When a child process is cloned, it shouldn't need to start decompiling methods it inherited. So as the method list gets larger, the CDC Java runtime environment may need a larger code cache to fit all the precompiled methods.

   To measure the code cache needed by the server JVM instance, perform the following procedure:

1. **Build a version of the CDC Java runtime environment withthe** `CVM_JIT_COLLECT_STATS=true` **build option.**

2. **Run** `cvm` **with** `-Xjit:stats=minimal` **option.**

   Be sure not to use a complicated GUI app to cause further method compilation.

   The statistics are displayed when the server JVM instance exits. For example,

   ```
   Allocated codebuffer bytes        :     656860      96     10868
   ```

   Once the size is found, run `cvm` with a codecache slightly above the actual use. In our case, `-Xjit:codeCacheSize=850K`.

12.6   Implementation Notes

The CDC AMS implementation is structured in two levels: the `mtask` extension to the CDC Java virtual machine and the AMS framework in `com.sun.appmanager`. The mtask extension is closely associated with Linux operating system process management. In principal, this layer could be adapted to another operating system, particularly one with `copy-on-write` process cloning semantics. However, the current implementation does not support easy porting to alternate operating systems.

At the AMS framework layer, the following areas represent the most accessible ways to adapt CDC AMS to a product implementation:

- **Presentation mode.** The sample presentation modes included with CDC AMS are meant to demonstrate application management functionality, not user experience. The presentation mode also acts as a registry for integrating different components of the CDC AMS framework.

- **Application model support.** The CDC AMS implementation includes support for the xlet application model. But other application models are possible.

- **Application support services.** The `preferences`, `store` and `apprepository` packages can be modified to support richer services.

- **OTA provisioning.** The CDC AMS implementation includes support for J2EE Client Provisioning (JSR-124). But the CDC AMS framework can be extended to support other OTA protocols.

- **Native AMS integration.** A native AMS could be integrated with CDC AMS through the presentation mode. `cvmc` can be used as an example for how to control JVM instances through the `mtask` command language.

Package
com.sun.appmanager.appmodel

Description

 The appmodel package encapsulates application models built on top of the mtask framework. The main abstraction is `com.sun.appmanager.AppModelController`, representing the lifecycle of a managed application as implemented on top of mtask. Currently only xlet's are supported as an application model, represented by `com.sun.appmanager.XletAppModelController`.

Class Summary	
Classes	
AppModelController$_{14}$	A class that encapsulates the lifecycle of a managed app model
XletAppModelController$_{16}$	A class that encapsulates the lifecycle of an xlet app model

com.sun.appmanager.appmodel

AppModelController

```
java.lang.Object
   |
   +--com.sun.appmanager.appmodel.AppModelController
```

Direct Known Subclasses: XletAppModelController₁₆

Declaration

public abstract class **AppModelController**

Description

A class that encapsulates the lifecycle of a managed app model

Member Summary	
Fields	
protected Client$_{34}$	mtaskClient$_{14}$
static int	XLET_APP_MODEL$_{14}$
Constructors	
	AppModelController$_{15}$(Client$_{34}$ mtaskClient)
Methods	
static AppModelController$_{14}$	getAppModelController$_{15}$(int appModelId, Client$_{34}$ client)

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Fields

mtaskClient

protected Client$_{34}$ **mtaskClient**

The basic mtask controller for this app model

XLET_APP_MODEL

public static final int **XLET_APP_MODEL**

Xlet application model, to be passed in to `getAppModelController()`.

Constructors

AppModelController(**Client**$_{34}$ mtaskClient)

    public **AppModelController**(Client$_{34}$ mtaskClient)

A new AppModelController with an underlying mtask client `mtaskClient`

Methods

getAppModelController(int appModelId, **Client**$_{34}$ client)

    public static AppModelController$_{14}$ **getAppModelController**(int appModelId, Client$_{34}$ client)

Create a new app model controller for a given app model id.

com.sun.appmanager.appmodel
XletAppModelController

```
java.lang.Object
   |
  +--AppModelController₁₄
         |
         +--com.sun.appmanager.appmodel.XletAppModelController
```

Declaration

public class **XletAppModelController** extends AppModelController₁₄

Description

A class that encapsulates the lifecycle of an xlet app model

Member Summary	
Fields	
static int	XLET_ACTIVE₁₇
static int	XLET_DESTROYED₁₇
static int	XLET_LOADED₁₇
static int	XLET_PAUSED₁₇
static int	XLET_UNKNOWN₁₇
Constructors	
	XletAppModelController₁₇(Client₃₄ mtaskClient)
Methods	
boolean	xletActivate₁₇(java.lang.String xletId)
boolean	xletDeactivate₁₇(java.lang.String xletId)
boolean	xletDestroy₁₈(java.lang.String xletId)
boolean	xletDestroy₁₈(java.lang.String xletId, boolean conditional)
int	xletGetState₁₈(java.lang.String xletId)
boolean	xletInitialize₁₈(java.lang.String xletId)
boolean	xletPause₁₈(java.lang.String xletId)
boolean	xletStart₁₈(java.lang.String xletId)
java.lang.String	xletStateToString₁₈(int state)

Inherited Member Summary
Fields inherited from class AppModelController₁₄
mtaskClient₁₄, XLET_APP_MODEL₁₄
Methods inherited from class AppModelController₁₄
getAppModelController(int, Client)₁₅

<table>
<tr><td>Inherited Member Summary</td></tr>
</table>

Inherited Member Summary

Methods inherited from class `Object`

```
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
toString(), wait(), wait(long), wait(long, int)
```

Fields

XLET_ACTIVE

   public static final int **XLET_ACTIVE**

XLET_DESTROYED

   public static final int **XLET_DESTROYED**

XLET_LOADED

   public static final int **XLET_LOADED**

XLET_PAUSED

   public static final int **XLET_PAUSED**

XLET_UNKNOWN

   public static final int **XLET_UNKNOWN**

Constructors

XletAppModelController(**Client**$_{34}$ mtaskClient)

   public **XletAppModelController**(Client$_{34}$ mtaskClient)

   Construct a new xlet controller based on an underlying mtask client.

Methods

xletActivate(**java.lang.String** xletId)

   public boolean **xletActivate**(java.lang.String xletId)

   Activate a target xlet: make it visible and cause its `startXlet()` method to be called.

xletDeactivate(**java.lang.String** xletId)

   public boolean **xletDeactivate**(java.lang.String xletId)

   De-activate a target xlet: make it invisible and cause its `pauseXlet()` method to be called.

xletDestroy(`java.lang.String` xletId)

    `public boolean `**`xletDestroy`**`(java.lang.String xletId)`

    Destroy target xlet `xletId` (cause its `destroyXlet()` method to be called).

xletDestroy(`java.lang.String` xletId, boolean conditional)

    `public boolean `**`xletDestroy`**`(java.lang.String xletId, boolean conditional)`

xletGetState(`java.lang.String` xletId)

    `public int `**`xletGetState`**`(java.lang.String xletId)`

    Get state of target xlet

xletInitialize(`java.lang.String` xletId)

    `public boolean `**`xletInitialize`**`(java.lang.String xletId)`

    Start target xlet `xletId` (cause its `initXlet()` method to be called).

xletPause(`java.lang.String` xletId)

    `public boolean `**`xletPause`**`(java.lang.String xletId)`

    Pause target xlet `xletId` (cause its `pauseXlet()` method to be called).

xletStart(`java.lang.String` xletId)

    `public boolean `**`xletStart`**`(java.lang.String xletId)`

    Start target xlet `xletId` (cause its `startXlet()` method to be called).

xletStateToString(int state)

    `public java.lang.String `**`xletStateToString`**`(int state)`

Package
com.sun.appmanager.apprepository

Description

The application repository holds persistent application related items. An instance of $AppRepository_{22}$ handles all available applications on the device. This includes default installed applications, or applications installed over the air.

Each locally installed application is represented by an instance of $AppModule_{20}$. This instance can be used to get information about the application.

Class Summary	
Interfaces	
$AppRepository_{22}$	Interface for the application manager's repository which contains necessary items like application files, application descriptors, etc.
Classes	
$AppModule_{20}$	

com.sun.appmanager.apprepository
AppModule

```
java.lang.Object
  |
  +--com.sun.appmanager.apprepository.AppModule
```

Declaration

public class **AppModule**

Member Summary	
Fields	
protected java.net.URL	iconDepressedURL$_{21}$
protected java.net.URL	iconURL$_{21}$
Constructors	
	AppModule$_{21}$(java.util.Hashtable hash)
Methods	
java.lang.String	getBundle$_{21}$()
java.lang.String	getIconDepressedPath$_{21}$()
java.net.URL	getIconDepressedURL$_{21}$()
java.lang.String	getIconPath$_{21}$()
java.net.URL	getIconURL$_{21}$()
java.lang.String	getInvocation$_{21}$()
boolean	getIsStartup$_{21}$()
boolean	getIsSwitchable$_{21}$()
java.lang.String	getMenu$_{21}$()
java.lang.String	getPath$_{22}$()
java.lang.String	getSecurityLevel$_{22}$()
java.lang.String	getTitle$_{22}$()
java.lang.String	getType$_{22}$()
boolean	isSublist$_{22}$()
boolean	isSystemApp$_{22}$()
java.lang.String	launch$_{22}$()
java.lang.String	toString$_{22}$()

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)

Fields

iconDepressedURL

```
protected java.net.URL iconDepressedURL
```

URL representation of the path to the depressed icon image

iconURL

```
protected java.net.URL iconURL
```

URL representation of the path to the icon image

Constructors

AppModule(`java.util.Hashtable` hash)

```
public AppModule(java.util.Hashtable hash)
```

Parameters:

`hash` - table containing AppModule properties

Methods

getBundle()

```
public java.lang.String getBundle()
```

Returns: The bundle this app belongs to. getIconDepressedPath()

```
public java.lang.String getIconDepressedPath()
```

Returns: path to the module's depressed icon image: getIconDepressedURL()

```
public java.net.URL getIconDepressedURL()
```

Returns: URL representation of the path to the module's depressed icon image: getIconPath()

```
public java.lang.String getIconPath()
```

Returns: path to the module's icon image: getIconURL()

```
public java.net.URL getIconURL()
```

Returns: URL representation of the path to the module's icon image: getInvocation()

```
public java.lang.String getInvocation()
```

Returns: printable version of the invocation command of the module: getIsStartup()

```
public boolean getIsStartup()
```

Returns: true if this is a startup application, false otherwise: getIsSwitchable()

```
public boolean getIsSwitchable()
```

Returns: true if a special-case ticker module, false otherwise: getMenu()

```
public java.lang.String getMenu()
```

Returns: name of the menu if this module represents a menu, null if the module does not represent a menu. getPath()

```
public java.lang.String getPath()
```

Returns: path to application files of the module: getSecurityLevel()

```
public java.lang.String getSecurityLevel()
```

Returns: The security level setting for this module: getTitle()

```
public java.lang.String getTitle()
```

Returns: title of application module: getType()

```
public java.lang.String getType()
```

Returns: type of application module: isSublist()

```
public boolean isSublist()
```

Returns: true if the module is a submenu, false otherwise: isSystemApp()

```
public boolean isSystemApp()
```

Returns: boolean if this app is a system app: launch()

```
public java.lang.String launch()
```

Launch the app and return a string handle

Returns: string representation of application id. toString()

```
public java.lang.String toString()
```

Overrides: `toString` in class `Object`

Returns: title and invocation command of the module: com.sun.appmanager.

apprepository

AppRepository

Declaration
```
public interface AppRepository
```

Description
Interface for the application manager's repository which contains necessary items like application files, application descriptors, etc.

Member Summary		
Methods		
AppModule$_{20}$[]	getAppList$_{23}$()	
Destination$_{57}$	getDestination$_{23}$(Descriptor$_{45}$ descriptor)	
AppModule$_{20}$[]	installApplication$_{23}$(Destination$_{57}$ destination, java.lang. String menu)	
boolean	removeApplication$_{23}$(AppModule$_{20}$[] module)	

Methods

getAppList()

    public AppModule$_{20}$[] **getAppList**()

Return an array of AppModules representing available applications on the device for the given user.

Returns: AppModule[] array of application modules for each available application and menu:

getDestination(`Descriptor`$_{45}$ descriptor)

    public Destination$_{57}$ **getDestination**(Descriptor$_{45}$ descriptor)

Create a Destination object to handle the download of a new application

Parameters:

`descriptor` - Descriptor object pertaining to the application to install

Returns: New Destination object, null if one cannot be created:

installApplication(`Destination`$_{57}$ destination, `java.lang.String` menu)

    public AppModule$_{20}$[] **installApplication**(Destination$_{57}$ destination, java.lang.String menu)

Install an app bundle containing one or more apps into the repository

Parameters:

`destination` - Destination object pertaining to the application to install

`menu` - the menu within the application manager in which this application should be installed in.

Returns: AppModule module object pertaining to the newly installed application, null the application could not be installed: removeApplication(`AppModule`$_{20}$[] module)

    public boolean **removeApplication**(AppModule$_{20}$[] module)

Remove an app bundle containing one or more apps from the repository

Parameters:

`module` - applications to be removed. These apps must all be part of the same app bundle.

Returns: boolean true if all apps were removed, false otherwise. This method makes an attempt to remove each application regardless of if any previous apps fail to be removed, but will return a false if any application fails to be removed completely. Also, all applications specified in module[] must be from the same bundle. This method will return false immediately if this is not the case.

c44eef8393e7952e

footer only

Package
com.sun.appmanager.client

Description

CDCAms client API's. These are the API's that are to be used by cdcams client xlets. `Client`$_{26}$ acts as a place holder for client API components. `com.sun.appmanager.client.ServiceManager` is responsible for allowing clients to register and access services.

Class Summary	
Classes	
`Client`$_{26}$	`Client` is the class that contains handles to each of the AppManager client modules.

com.sun.appmanager.client
Client

```
java.lang.Object
  |
  +--com.sun.appmanager.client.Client
```

Declaration
public abstract class **Client**

Description
Client is the class that contains handles to each of the AppManager client modules. Only one Client instance is created per cdcams client. The static methods of Client may be used to get access to client components.

Member Summary	
Fields	
protected java.lang. String	clientId$_{26}$
Constructors	
	Client$_{27}$()
Methods	
static java.lang. String	getClientId$_{27}$()

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Fields

clientId

protected java.lang.String **clientId**

The id of this client instance, as identified by mtask

Constructors

Client()

    public **Client**()

Create a singleton instance.

If more than one instance is created, a RuntimeException is thrown.

Methods

getClientId()

    public static java.lang.String **getClientId**()

Return the mtask client id of this client

Package
com.sun.appmanager

Description

The root package contains the AppManager$_{30}$ abstract class. Customers adopting AppManager need to subclass this class to start up their system and instantiate their module implementations. There can only be one AppManager$_{30}$ instance in the system. Static methods in AppManager$_{30}$ are designed to return handles to each AppManager module implementation.

Class Summary	
Classes	
AppManager$_{30}$	AppManager is the superclass that contains handles to each of the AppManager modules.

com.sun.appmanager

AppManager

```
java.lang.Object
   |
   +--com.sun.appmanager.AppManager
```

Declaration

`public abstract class AppManager`

Description

`AppManager` is the superclass that contains handles to each of the AppManager modules. Only one `AppManager` instance can be created. Subsequent instances cause a `RuntimeException`

Member Summary	
Fields	
protected static java. util.ResourceBundle	appManagerRB$_{31}$
protected static AppRepository$_{22}$	appRepository$_{31}$
protected static Client$_{34}$	mtaskClient$_{31}$
protected static OTAFactory$_{68}$	otaFactory$_{31}$
protected static PersistentStore$_{82}$	persistentStore$_{31}$
protected static Preferences$_{74}$	preferences$_{31}$
protected static PresentationMode$_{79}$	presentationMode$_{31}$
protected static java. rmi.registry.Registry	registry$_{31}$
Constructors	
	AppManager$_{31}$()
Methods	
static AppRepository$_{22}$	getAppRepository$_{32}$()
static Client$_{34}$	getMtaskClient$_{32}$()
static OTAFactory$_{68}$	getOTAFactory$_{32}$()
static PersistentStore$_{82}$	getPersistentStore$_{32}$()
static Preferences$_{74}$	getPreferences$_{32}$()
static PresentationMode$_{79}$	getPresentationMode$_{32}$()
static java.rmi. registry.Registry	getRegistry$_{32}$()
static java.util. ResourceBundle	getResourceBundle$_{32}$()

Inherited Member Summary

Methods inherited from class `Object`

```
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
toString(), wait(), wait(long), wait(long, int)
```

Fields

appManagerRB

protected static java.util.ResourceBundle **appManagerRB**

appRepository

protected static AppRepository$_{22}$ **appRepository**

mtaskClient

protected static Client$_{34}$ **mtaskClient**

otaFactory

protected static OTAFactory$_{68}$ **otaFactory**

persistentStore

protected static PersistentStore$_{82}$ **persistentStore**

preferences

protected static Preferences$_{74}$ **preferences**

presentationMode

protected static PresentationMode$_{79}$ **presentationMode**

registry

protected static java.rmi.registry.Registry **registry**

Constructors

AppManager()

public **AppManager**()

Create a singleton instance.

If more than one instance is created, a RuntimeException is thrown.

Methods

getAppRepository()

    public static AppRepository₂₂ **getAppRepository**()

getMtaskClient()

    public static Client₃₄ **getMtaskClient**()

getOTAFactory()

    public static OTAFactory₆₈ **getOTAFactory**()

getPersistentStore()

    public static PersistentStore₈₂ **getPersistentStore**()

getPreferences()

    public static Preferences₇₄ **getPreferences**()

getPresentationMode()

    public static PresentationMode₇₉ **getPresentationMode**()

getRegistry()

    public static java.rmi.registry.Registry **getRegistry**()

getResourceBundle()

    public static java.util.ResourceBundle **getResourceBundle**()

Package
com.sun.appmanager.mtask

Description

The Client$_{34}$ class is used to communicate to a *master JVM instance* responsible for replicating itself to create new JVM instances. All aspects of application lifecycle can be implemented via the Client$_{34}$ class—-starting, messaging, listing and killing applications.

The master JVM warmup activities can also be controlled via this class, by using the warmupLists() method.

Class Summary	
Interfaces	
TaskListener$_{38}$	
Classes	
Client$_{34}$	A class that encapsulates communication with a *master JVM instance*.

com.sun.appmanager.mtask
Client

```
java.lang.Object
  |
  +--com.sun.appmanager.mtask.Client
```

Declaration
public class **Client**

Description
A class that encapsulates communication with a *master JVM instance*. The system-wide instance of this class can be obtained via the getMtaskClient() method of class AppManager$_{30}$. This instance can control all aspects of application lifecycle.

Member Summary	
Fields	
static java.lang. String	DEFAULT_SERVER_NAME$_{35}$
static int	DEFAULT_SERVER_PORT$_{35}$
Constructors	
	Client$_{35}$()
	Client$_{35}$(int port)
	Client$_{35}$(java.lang.String server)
	Client$_{35}$(java.lang.String server, int port)
Methods	
void	addListener$_{35}$(TaskListener$_{38}$ l)
boolean	broadcast$_{36}$(java.lang.String messageWord)
boolean	exit$_{36}$()
java.lang.String[]	getLaunchedApps$_{36}$()
boolean	kill$_{36}$(java.lang.String appHandle)
boolean	killall$_{36}$()
java.lang.String	launch$_{36}$(java.lang.String command, java.lang.String[] args)
java.lang.String[]	list$_{36}$()
boolean	message$_{36}$(java.lang.String appHandle, java.lang.String messageWord)
java.lang.String	messageWithResponse$_{36}$(java.lang.String appHandle, java.lang. String messageWord)
void	removeListener$_{36}$(TaskListener$_{38}$ l)
boolean	setenv$_{37}$(java.lang.String keyValuePair)
void	source$_{37}$(java.lang.String[] args)
boolean	testingMode$_{37}$(java.lang.String filePrefix)
void	warmupLists$_{37}$(java.lang.String classPath, java.lang.String classNames, java.lang.String memberNames)

Fields

DEFAULT_SERVER_NAME

 public static final java.lang.String **DEFAULT_SERVER_NAME**

DEFAULT_SERVER_PORT

 public static final int **DEFAULT_SERVER_PORT**

Constructors

Client(`java.lang.String` server, int port)

 public **Client**(java.lang.String server, int port)

 Create a client that communicates with server:port

Client(int port)

 public **Client**(int port)

 Create a client that communicates with <DEFAULT_SERVER_NAME>:port

Client(`java.lang.String` server)

 public **Client**(java.lang.String server)

 Create a client that communicates with server:<DEFAULT_SERVER_PORT>

Client()

 public **Client**()

 Create a client that communicates with <DEFAULT_SERVER_NAME>:<DEFAULT_SERVER_PORT>

Methods

addListener(**TaskListener**$_{38}$ l)

 public void **addListener**(TaskListener$_{38}$ l)

 Add task listener for task events

broadcast(`java.lang.String` messageWord)

```
public boolean broadcast(java.lang.String messageWord)
```

Broadcast message to all running apps

exit()

```
public boolean exit()
```

Tell the master JVM to exit

getLaunchedApps()

```
public java.lang.String[] getLaunchedApps()
```

kill(`java.lang.String` appHandle)

```
public boolean kill(java.lang.String appHandle)
```

Kill application

killall()

```
public boolean killall()
```

Kill all running applications

launch(`java.lang.String` command, `java.lang.String[]` args)

```
public java.lang.String launch(java.lang.String command, java.lang.String[] args)
```

Launch using the right command.

list()

```
public java.lang.String[] list()
```

Return a list of active applications

message(`java.lang.String` appHandle, `java.lang.String` messageWord)

```
public boolean message(java.lang.String appHandle, java.lang.String messageWord)
```

Send a message to an application

messageWithResponse(`java.lang.String` appHandle, `java.lang.String` messageWord)

```
public java.lang.String messageWithResponse(java.lang.String appHandle, java.lang.String
        messageWord)
```

Send a message to an application and expect a response

removeListener(`TaskListener`$_{38}$ l)

```
public void removeListener(TaskListener38 l)
```

Remove task listener for task events

setenv(`java.lang.String` keyValuePair)

    public boolean **setenv**(java.lang.String keyValuePair)

Set environment variable in the mTASK server. Subsequent children will inherit this.

source(`java.lang.String[]` args)

    public void **source**(java.lang.String[] args)

Source an arbitrary command in the mtask server

testingMode(`java.lang.String` filePrefix)

    public boolean **testingMode**(java.lang.String filePrefix)

Put mtask in testing mode where output, error and exit codes are dumped into files with prefix

warmupLists(`java.lang.String` classPath, `java.lang.String` classNames, `java.lang.String` memberNames)

    public void **warmupLists**(java.lang.String classPath, java.lang.String classNames, java.
            lang.String memberNames)

Warmup the running master JVM server.

com.sun.appmanager.mtask

TaskListener

Declaration

```
public interface TaskListener
```

Member Summary	
Fields	
static int	CDCAMS_TASK_KILLED_{38}
Methods	
void	taskEvent_{38}(java.lang.String appId, int what)

Fields

CDCAMS_TASK_KILLED

```
public static final int CDCAMS_TASK_KILLED
```

Methods

taskEvent(`java.lang.String` appId, int what)

```
public void taskEvent(java.lang.String appId, int what)
```

Package
com.sun.appmanager.ota

Description

Provides the classes and interfaces for over-the-air provisioning operations. In order to create an instance of an OTA object which understands a certain protocol, provisioning client software such as a Discovery Application (DA) can use the AppManager$_{32}$ method to retrieve the current AppManager's OTAFactory$_{68}$, and then use the factory to create an OTA object:

```
OTAFactory otaFactory = MyAppManager.getOTAFactory();
OTA omaOTA = otaFactory.getImpl( "OMA" );
```

The protocol specified in the OTAFactory$_{68}$ call is used to find an implementation class which supports the requested protocol. Alternatively, a class identifier can be passed to the OTAFactory$_{68}$ method, in which case the class indicated should have a constructor with no arguments.

Once an OTA implementation object is created, it can be used to discover downloads available at a given URL, which are returned in a Hashtable which contains objects dependent on the OTA implementation used.

```
Hashtable h = omaOTA.discover( "dummyURL" );
```

From this Hashtable, the DA can determine a URI for a download descriptor containing information about a particular download, which the OTA implementation will contact to populate a Descriptor object.

```
Descriptor des = omaOTA.createDescriptor( "dummyURI" );
```

Normally this descriptor will contain enough information to determine whether or not the download should continue, along with a URI to contact for the actual download data. The DA can then perform the actual download, providing the OTA implementation with enough information to allow it to pass the downloaded data to the DA, and to update the DA with download progress.

```
boolean result = omaOTA.download( des,
                        destinationObject,
                        dlIndicatorObject );
```

The actual downloaded data will be delivered in chunks to the destinationObject to interpret as it wishes. The destinationObject will be responsible for allocating necessary memory to hold downloaded data, and for saving it if desired.

As can be inferred from the example above, a Descriptor$_{45}$ object contains information about a particular download. A DA with a graphical interface can implement the DLIndicator$_{60}$ interface to be notified as a download progresses. The DA should implement the Destination$_{57}$ interface to provide the OTA implementation with information about what to do with downloaded data.

Downloaded executable data will be represented by an custom extension of the Application$_{41}$ class, which helps identify what type of application has been retrieved and how to invoke it properly.

Class Summary
Interfaces

Class Summary	
Destination$_{57}$	Generic interface to be used to specify different ways to store downloaded data from OTA implementation.
DLIndicator$_{60}$	Provides an interface for a download client to be informed regarding data download progress.
Classes	
Application$_{41}$	A representation of downloaded executable application content.
Descriptor$_{45}$	This class describes downloadable content.
OTA$_{62}$	Abstract class specifying the provisioning client agent.
OTAFactory$_{68}$	Abstract class providing a factory-like source for OTA download agents.
Exceptions	
OTAException$_{67}$	
OTANotSupportedException$_{69}$	
SyntaxException$_{71}$	Indicates an error encountered while parsing a download descriptor.

com.sun.appmanager.ota
Application

```
java.lang.Object
  |
  +--com.sun.appmanager.ota.Application
```

Declaration
public abstract class **Application**

Description
A representation of downloaded executable application content. Because we can have different app types (e.g., XLET v. MIDLET), use this class to hold relevant information needed to start an application properly.

Member Summary	
Fields	
protected int	appType₄₂
static int	MAIN_APP₄₂
static int	MIDLET_APP₄₂
static int	XLET_APP₄₂
Constructors	
	Application₄₂(java.lang.String name, java.lang.String iconPath, java.lang.String clazz, java.lang.String classpath)
Methods	
void	addProperty₄₂(java.lang.String name, java.lang.String value)
void	check₄₃()
int	getAppType₄₃()
java.lang.String	getClasspath₄₃()
java.lang.String	getIconPath₄₃()
java.lang.String	getMainClass₄₃()
java.lang.String	getName₄₃()
void	setIconPath₄₃(java.lang.String path)
void	setName₄₃(java.lang.String name)

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Fields

appType

```
protected int appType
```

MAIN_APP

```
public static int MAIN_APP
```

MIDLET_APP

```
public static int MIDLET_APP
```

XLET_APP

```
public static int XLET_APP
```

Constructors

Application(`java.lang.String` name, `java.lang.String` iconPath, `java.lang.String` clazz, `java.lang.String` classpath)

```
public Application(java.lang.String name, java.lang.String iconPath, java.lang.String
             clazz, java.lang.String classpath)
```

Create an instance of an application.

Parameters:

name - **The application's name**

iconPath - **The location of the application's icon**

clazz - **The main class of the application.**

Methods

addProperty(`java.lang.String` name, `java.lang.String` value)

```
public void addProperty(java.lang.String name, java.lang.String value)
             throws SyntaxException
```

Add a key/value pair to the application's list of properties.

Parameters:

name - - A key value.

value - - To be associated with the key.

Throws:

SyntaxException$_{71}$ - - If either name or value is **null**.

check()

```
public void check()
               throws SyntaxException
```

Validate the application and make sure that it is internally consistent.

Throws:

SyntaxException$_{71}$ - In case of inconsistency.

getAppType()

```
public int getAppType()
```

Determine the type of this application.

Returns: One of Application's defined application types, MAIN_APP, XLET_APP, or MIDLET_APP.

getClasspath()

```
public java.lang.String getClasspath()
```

Get the applications's specified classpath

Returns: A String indicating the application's specified classpath, or null if one is not specified:

getIconPath()

```
public java.lang.String getIconPath()
```

Get the path to the application's icon.

Returns: A String defining the path to the icon in the downloaded content. getMainClass()

```
public java.lang.String getMainClass()
```

Get the application's main class.

Returns: A String indicating the application's main class, or null if there is no main class.

getName()

```
public java.lang.String getName()
```

Get the application's name.

Returns: The application's name. setIconPath(java.lang.String path)

```
public void setIconPath(java.lang.String path)
```

Set the path to the application's icon.

setName(java.lang.String name)

```
public void setName(java.lang.String name)
```

Set the application's name.

com.sun.appmanager.ota
Descriptor

```
java.lang.Object
   |
   +--com.sun.appmanager.ota.Descriptor
```

All Implemented Interfaces: `java.io.Serializable`

Declaration

`public class Descriptor implements java.io.Serializable`

Description

This class describes downloadable content. This class is populated during descriptor parsing. Upon the completion of parsing, the $Descriptor_{50}$ method must be called to verify the consistency of the descriptor. If $Descriptor_{50}$ succeeds (i.e. doesn't throw an exception), then the descriptor must be taken as valid.

Member Summary	
Fields	
java.util.Vector	$applications_{47}$
protected int	$appManagerType_{47}$
protected java.lang. String	$classpath_{47}$
protected java.lang. String	$data_{47}$
java.util.Vector	$dependencies_{47}$
protected java.lang. String	$description_{47}$
protected java.lang. String	$displayName_{47}$
protected java.lang. String	$href_{47}$
protected java.lang. String	$iconURI_{48}$
protected java.lang. String	$installNotifyURI_{48}$
protected boolean	$isLibrary_{48}$
protected boolean	$isNative_{48}$
protected java.lang. String	$name_{48}$
protected java.lang. String	$objectURI_{48}$
protected java.lang. String	$schema_{48}$
protected java.lang. String	$securityLevel_{48}$
protected int	$size_{48}$

Member Summary

protected java.lang. String	source48
protected java.lang. String	type49
static int	TYPE_APP49
static int	TYPE_DATA49
static int	TYPE_LIB49
protected java.lang. String	vendor49
protected java.lang. String	version49

Constructors

	Descriptor49(java.lang.String schema, java.lang.String source)

Methods

void	addApplication49(Application41 app)
void	addData50(java.lang.String mimeType, java.lang.String href)
void	addLibrary50(boolean java)
void	checkOut50()
protected static void	checkURL50(java.lang.String uri)
java.util.Vector	getApplications51()
java.lang.String	getDataRef51()
java.lang.String	getDescription51()
java.lang.String	getDisplayName51()
java.lang.String	getIconURI51()
java.lang.String	getInstallNotifyURI51()
java.lang.String	getMimeType51()
java.lang.String	getName51()
java.lang.String	getObjectURI51()
java.lang.String	getSecurityLevel52()
int	getSize52()
java.lang.String	getSource52()
java.lang.String	getVendor52()
java.lang.String	getVersion52()
boolean	isData52()
boolean	isJavaLibrary52()
boolean	isNativeLibrary52()
void	setAppManagerType52(int type)
void	setDescription53(java.lang.String description)
void	setDisplayName53(java.lang.String s)
void	setIconURI53(java.lang.String uri)
void	setInstallNotifyURI53(java.lang.String uri)
void	setName54(java.lang.String name)
void	setObjectURI54(java.lang.String uri)
void	setSecurityLevel54(java.lang.String level)
void	setSize55(int size)
void	setSize54(java.lang.String size)
void	setType55(java.lang.String type)
void	setVendor55(java.lang.String vendor)
void	setVersion55(java.lang.String version)

Inherited Member Summary
Methods inherited from class `Object`
`clone()`, `equals(Object)`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()`, `wait()`, `wait(long)`, `wait(long, int)`

Fields

applications

`public java.util.Vector` **`applications`**

A vector of applications contained in this media object.

appManagerType

`protected int` **`appManagerType`**

The Appmanager type of the media object. Normally Appmanager treats downloads as either applications $Descriptor_{49}$, libraries $Descriptor_{49}$ or binary data of some sort $Descriptor_{49}$.

classpath

`protected java.lang.String` **`classpath`**

For application media objects, a classpath.

data

`protected java.lang.String` **`data`**

The mimetype of a data object.

dependencies

`public java.util.Vector` **`dependencies`**

A vector of dependencies upon which this media object relies.

description

`protected java.lang.String` **`description`**

A short, textual description of the media object. There are no particular semantics associated with the description, but it should be displayed to the user prior to installation.

displayName

`protected java.lang.String` **`displayName`**

A displayable name of this media object for use on the device, if desired.

href

`protected java.lang.String` **`href`**

The URI of a data object.

iconURI

```
protected java.lang.String iconURI
```

The URI of an icon object which can be used by the client to represent the media object.

installNotifyURI

```
protected java.lang.String installNotifyURI
```

A URI to notify after a successful download and installation.

isLibrary

```
protected boolean isLibrary
```

An indicator of whether the media object is a library.

isNative

```
protected boolean isNative
```

An indicator of whether the media object is a native executable.

name

```
protected java.lang.String name
```

The user-readable name of the media object, which identifies it to the user.

objectURI

```
protected java.lang.String objectURI
```

The URI (usually URL) from which the object can be downloaded. This must be accessible via an HTTP GET in order to allow the client agent to perform a download.

schema

```
protected java.lang.String schema
```

securityLevel

```
protected java.lang.String securityLevel
```

The security level which the Appmanager will associate with this media object.

size

```
protected int size
```

source

```
protected java.lang.String source
```

type

```
protected java.lang.String type
```

The MIME type of the media object.

TYPE_APP

```
public static int TYPE_APP
```

An identifier used to denote a media object which is an executable application.

TYPE_DATA

```
public static int TYPE_DATA
```

An identifier used to denote a media object which is binary data of some sort (i.e., not executable).

TYPE_LIB

```
public static int TYPE_LIB
```

An identifier used to denote a media object which is a library object for use by other applications.

vendor

```
protected java.lang.String vendor
```

The organization which provides the media object.

version

```
protected java.lang.String version
```

The version of this descriptor.

Constructors

Descriptor(**java.lang.String** schema, **java.lang.String** source)

```
public Descriptor(java.lang.String schema, java.lang.String source)
```

Methods

addApplication(**Application**$_{41}$ app)

```
public void addApplication(Application41 app)
```

Add a ddx:application entry to the list of applications in this media object.

Parameters:

`app` - An `Application`₄₁ object constructed from information provided in the application's entry.

addData(`java.lang.String` mimeType, `java.lang.String` href)

```
public void addData(java.lang.String mimeType, java.lang.String href)
             throws SyntaxException
```

For a data object, record a mimetype and href.

Parameters:

`mimeType` - A MIME type for the data

`href` - An href indicating where to find the data (may be null, in which case the data will be assumed to be local).

Throws:

`SyntaxException`₇₁ - if data has already been set, or if there is an apparent conflict in the download type.

addLibrary(boolean java)

```
public void addLibrary(boolean java)
             throws SyntaxException
```

Indicate that this media object contains a library.

Parameters:

`java` - Whether or not the library is Java (as opposed to native).

Throws:

`SyntaxException`₇₁ - if we have already recorded a data or application media object.

checkOut()

```
public void checkOut()
             throws SyntaxException
```

Validate the descriptor by verifying that all of the necessary fields have been populated, and that no contradictory information was included. As part of the process, each `Application`₄₁ identified as part of the media object is also checked for internal consistency.

Throws:

`SyntaxException`₇₁ - If mandantory fields are missing, the descriptor contains contradictory information, or any of the Applications in the media object are internally inconsistent.

checkURL(`java.lang.String` uri)

```
protected static void checkURL(java.lang.String uri)
             throws SyntaxException
```

Check a provided URL or URI for valid syntax.

Parameters:

uri - **The URI or URI to check.**

Throws:

[SyntaxException](71) - If there is a problem with the URI.

getApplications()

```
public java.util.Vector getApplications()
```

Return a Vector containing all of the Application$_{41}$ objects from this media object. If this is not an application download, the Vector will be null.

getDataRef()

```
public java.lang.String getDataRef()
```

Return the data href (URI) for this media object. If this is not a data download, the href will be null.

getDescription()

```
public java.lang.String getDescription()
```

Return the description of this media object (possibly null).

getDisplayName()

```
public java.lang.String getDisplayName()
```

Retrieve the name of this media object for display. If the display name is not set, return the media object's name sttribute.

getIconURI()

```
public java.lang.String getIconURI()
```

Return the URI of this media object's icon (may be null).

getInstallNotifyURI()

```
public java.lang.String getInstallNotifyURI()
```

Return the URI of this media object (possibly null) which should be contacted after installation.

getMimeType()

```
public java.lang.String getMimeType()
```

Return the MIME type of this media object.

getName()

```
public java.lang.String getName()
```

Return the name of this media object (possibly null).

getObjectURI()

```
public java.lang.String getObjectURI()
```

Return the URI of this media object to be used for actual download.

getSecurityLevel()

```
public java.lang.String getSecurityLevel()
```

Return the security level of this media object, possibly null.

getSize()

```
public int getSize()
```

Return the size (a non-negative value) of the media object for download.

getSource()

```
public java.lang.String getSource()
```

getVendor()

```
public java.lang.String getVendor()
```

Retrieve the vendor of this media object (may be null).

getVersion()

```
public java.lang.String getVersion()
```

Retrieve the version of this media object (may be null).

isData()

```
public boolean isData()
```

Indicate whether or not this media object is binary data.

isJavaLibrary()

```
public boolean isJavaLibrary()
```

Indicate whether or not this media object is a Java library.

isNativeLibrary()

```
public boolean isNativeLibrary()
```

Indicate whether or not this media object is a native library.

setAppManagerType(int type)

```
public void setAppManagerType(int type)
            throws SyntaxException
```

Set the Appmanager type of this media object. Currently downloads recognized by Appmanager are either applications, libraries or data downloads.

Parameters:

`type` - The Appmanager type

Throws:

`SyntaxException`$_{71}$ - if the Appmanager type has already been set.

setDescription(`java.lang.String` description)

```
public void setDescription(java.lang.String description)
            throws SyntaxException
```

Set the description for the media object.

Parameters:

`description` - The new description string.

Throws:

`SyntaxException`$_{71}$ - if the description has already been set.

setDisplayName(`java.lang.String` s)

```
public void setDisplayName(java.lang.String s)
```

Set the display name of this download.

Parameters:

`s` - The name for display use to identify the download.

setIconURI(`java.lang.String` uri)

```
public void setIconURI(java.lang.String uri)
            throws SyntaxException
```

Set the iconURI for this media object.

Parameters:

`uri` - The URI (URL) which will provide an icon which will provide an icon for this media object.

Throws:

`SyntaxException`$_{71}$ - If the URI is invalid.

setInstallNotifyURI(`java.lang.String` uri)

```
public void setInstallNotifyURI(java.lang.String uri)
            throws SyntaxException
```

Set the installNotifyURI for this media object. This URI should be notified once download and installation are complete.

Parameters:

uri - **The URI to contact**

Throws:

SyntaxException₇₁ - **if the URI is invalid.**

setName(`java.lang.String` name)

```
public void setName(java.lang.String name)
             throws SyntaxException
```

Set the name of the download object.

Parameters:

name - **The name to use for this descriptor.**

Throws:

SyntaxException₇₁ - **if the name is already set.**

setObjectURI(`java.lang.String` uri)

```
public void setObjectURI(java.lang.String uri)
             throws SyntaxException
```

Set the objectURI for this media object, from which the download may be performed.

Parameters:

uri - **The URI (URL) to contact.**

Throws:

SyntaxException₇₁ - **if the objectURI has already been set, or if the parameter is an invalid URL.**

setSecurityLevel(`java.lang.String` level)

```
public void setSecurityLevel(java.lang.String level)
             throws SyntaxException
```

Set the security level of this media object. Security levels recognized by Appmanager are
`SecurityConstrained` and `SecurityTrusted`.

Parameters:

level - **The security level to set**

Throws:

SyntaxException₇₁ - **if the specified security level is unknown.**

setSize(`java.lang.String` size)

```
public void setSize(java.lang.String size)
             throws SyntaxException
```

Set the size of this download.

Parameters:

`size` - The download size in bytes.

Throws:

[SyntaxException](71) - if the size has already been set, or if the size value cannot be properly parsed into a non-negative, integer value.

setSize(int size)

```
public void setSize(int size)
                throws SyntaxException
```

Set the size of this download.

Parameters:

`size` - The download size in bytes.

Throws:

[SyntaxException](71) - if the size has already been set, or if the size value is not a non-negative value.

setType(`java.lang.String` type)

```
public void setType(java.lang.String type)
                throws SyntaxException
```

Set the type of this download.

Parameters:

`type` - The download type.

Throws:

[SyntaxException](71) - if the type has already been set.

setVendor(`java.lang.String` vendor)

```
public void setVendor(java.lang.String vendor)
                throws SyntaxException
```

Set the vendor which provided this media object.

Parameters:

`vendor` - The vendor identifier.

Throws:

SynxtaxException - If the vendor has already been set.
    SyntaxException[71]

setVersion(`java.lang.String` version)

```
public void setVersion(java.lang.String version)
                throws SyntaxException
```

Set the version of this download.

Parameters:

`version` - The version of this download.

Throws:

SyntaxException$_{71}$ - if the version is already set.

com.sun.appmanager.ota
Destination

Declaration

```
public interface Destination
```

Description

Generic interface to be used to specify different ways to store downloaded data from OTA implementation. An instance of destination class should be provided to the OTA_{65} method. Since there are few methods that practically change the state of a destination, this class should not be MT safe, and multiple instances should be used for each call to OTA_{65}.

Member Summary	
Methods	
void	$abort_{57}$()
void	$acceptMimeType_{57}$(java.lang.String mimeType)
void	$finish_{58}$()
int	$getMaxChunkSize_{58}$()
int	$receive_{58}$(java.io.InputStream in, int desiredLength)
void	$start_{58}$(java.lang.String sourceURL, java.lang.String mimeType)

Methods

abort()

```
public void abort()
```

Notifies the destination that the object acquisition has failed or was stopped. There will be no more bits sent, and the $Destination_{58}$ method will not be called.

acceptMimeType(java.lang.String mimeType)

```
public void acceptMimeType(java.lang.String mimeType)
          throws OTAException
```

Asks a destination whether it supports a certain mime type.

Parameters:

`mimeType` - mimeType to be sent to this destination.

Throws:

OTAException$_{67}$ - if this mime type is not supported.

finish()

```
public void finish()
              throws OTAException, IOException
```

Notifies the destination that the object download is done.

Throws:

OTAException$_{67}$ - it is at implemntation's discretion to throw an OTAException if anything goes
      wrong. The download will be cancelled and this exception will be bubbled up.

`java.io.IOException` - signal that there was an IO error.

getMaxChunkSize()

```
public int getMaxChunkSize()
```

Asks the destination what's the maximum buffer size should be used for downloading.

**Returns: maximum buffer size to be used for downloading, or 0, to let the OTA implementation
      decide what's best.
      : receive(`java.io.InputStream` in, int desiredLength)**

```
public int receive(java.io.InputStream in, int desiredLength)
              throws OTAException, IOException
```

Requests this destination to receive a part of the object being downloaded.

Parameters:

`in` - The InputStream to read from

`desiredLength` - The number of bytes to receive

**Returns: the total number of bytes read, or −1 is there is no more data because the end of the stream
      has been reached. Throws:**

OTAException$_{67}$ - it is at implemntation's discretion to throw an OTAException if anything goes
      wrong. The download will be cancelled and this exception will be bubbled up.

`java.io.IOException` - signal that there was an IO error.

start(`java.lang.String` sourceURL, `java.lang.String` mimeType)

```
public void start(java.lang.String sourceURL, java.lang.String mimeType)
              throws OTAException, IOException
```

Notifies the destination the download is about to start.

Parameters:

`sourceURL` - source URL

`mimeType` - mime type of the object to be sent to this destination.

Throws:

`OTAException`$_{67}$ - it is at implemntation's discretion to throw an OTAException if anything goes wrong. The download will be cancelled and this exception will be bubbled up.

`java.io.IOException` - signal that there was an IO error.

com.sun.appmanager.ota
DLIndicator

Declaration

```
public interface DLIndicator
```

Description

Provides an interface for a download client to be informed regarding data download progress.

Member Summary	
Methods	
void	downloadDone$_{60}$()
int	getGranularity$_{60}$()
java.lang.Object	getLockNotifier$_{60}$()
boolean	isCancelled$_{60}$()
void	updatePercent$_{61}$(int value)

Methods

downloadDone()

```
public void downloadDone()
```

Notify that the download is done. This doesn't mean, however, that the download is successfully completed or 100% has been reached.

getGranularity()

```
public int getGranularity()
```

Called once before download is about to start. The value returns tells how frequently the indicator should be updated in percents. So, returning 1 will cause download to update indicator every time one more percent is downloaded.

Returns: granularity: getLockNotifier()

```
public java.lang.Object getLockNotifier()
```

Return a synchronization object for download. The object returned from this method will be used by download thread to wait on so download is either completed by the separate download thread, or is cancelled by the system. So, to cancel the download, one must call notify on the same object and then make isCancelled$_{60}$ to return true. If you return null from this method, the cancellation will still be possible, but if IO is waiting, it won't happen until it's done waiting.

Returns: object to syncrhonize download and cancellation: isCancelled()

```
public boolean isCancelled()
```

Check for cancelled download. If at any time this method is called it returns true, download is cancelled.

Returns: true if user/system cancels the download. updatePercent(int value)

`public void `**`updatePercent`**`(int value)`

Update callback. Called by downloaded each time a certain amount of data has been downloaded.

Parameters:

`value` - how much data in percentage is downloaded

com.sun.appmanager.ota
OTA

```
java.lang.Object
  |
  +--com.sun.appmanager.ota.OTA
```

Declaration

public abstract class **OTA**

Description

Abstract class specifying the provisioning client agent.

Member Summary	
Fields	
static java.lang.String	ST_ATTRIBUTEMISMATCH$_{63}$
static java.lang.String	ST_DEVICEABORTED$_{63}$
static java.lang.String	ST_INSUFFICIENTMEMORY$_{63}$
static java.lang.String	ST_INVALIDDDVERSION$_{63}$
static java.lang.String	ST_INVALIDDESCRIPTOR$_{63}$
static java.lang.String	ST_LOADERERROR$_{64}$
static java.lang.String	ST_LOSSOFSERVICE$_{64}$
static java.lang.String	ST_NONACCEPTABLECONTENT$_{64}$
static java.lang.String	ST_SIZEMISMATCH$_{64}$
static java.lang.String	ST_SUCCESS$_{64}$
static java.lang.String	ST_USERCANCELLED$_{64}$
Constructors	
	OTA$_{64}$()
Methods	
Descriptor$_{45}$	createDescriptor$_{65}$(java.lang.String uri)
java.util.Hashtable	discover$_{65}$(java.lang.String url)
boolean	download$_{65}$(Descriptor$_{45}$ des, Destination$_{57}$ store, DLIndicator$_{60}$ report)
abstract java.lang.String	getSchema$_{65}$()

Member Summary	
boolean	sendNotify$_{66}$(java.lang.String uri, java.lang.String statusCode, java.lang.String statusMsg)

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Fields

ST_ATTRIBUTEMISMATCH

public static final java.lang.String **ST_ATTRIBUTEMISMATCH**

Status constant that indicates that the media object does not match the attributes defined in the Download Descriptor, and that the device therefore rejects the media object.

ST_DEVICEABORTED

public static java.lang.String **ST_DEVICEABORTED**

Status constant that indicates that the device interrupted, or cancelled, the retrieval of the medai object despite the fact that the content should have been executable on the device. This is thus a different case from "Insuficient Memory" and "Non-Acceptable Content" (where the device has concluded that it cannot use the content).

ST_INSUFFICIENTMEMORY

public static final java.lang.String **ST_INSUFFICIENTMEMORY**

Status constant that indicates to the service that the device could not accept the media object as it did not have enough storage space on the device. This event may occur before ar oafter the retrieval of the media object.

ST_INVALIDDDVERSION

public static final java.lang.String **ST_INVALIDDDVERSION**

Status constant that indicates that the device was not compatible with the "major" version of the Download Descriptor, as indicated in the Attribute Version (that is a parameter to the attribute Media).

ST_INVALIDDESCRIPTOR

public static final java.lang.String **ST_INVALIDDESCRIPTOR**

Status constant that indicates that the device could not interpret the Download Descriptor. This typically means a syntactic error.

ST_LOADERERROR

```
public static java.lang.String ST_LOADERERROR
```

Status constant that indicates that the URL that was to be used for the retrieval of the Medai Object did not provide access to the Media Object. This may represent for example errors of type server down, incorrect URL and service errors.

ST_LOSSOFSERVICE

```
public static final java.lang.String ST_LOSSOFSERVICE
```

Status constant that indicates the the client device lost network service while retrieveing the Media Object.

ST_NONACCEPTABLECONTENT

```
public static java.lang.String ST_NONACCEPTABLECONTENT
```

Status constant that indicates that after the retrieval of the media object, buf before sedning the installtion notification, the Download Agent concluded that the device cannot use the media object.

ST_SIZEMISMATCH

```
public static final java.lang.String ST_SIZEMISMATCH
```

Statis constant that indicates that the downloaded media object size was not the same as stated in the descriptor.

ST_SUCCESS

```
public static final java.lang.String ST_SUCCESS
```

Status constant that indicates to the service that the media object was downloaded and installed successfully.

ST_USERCANCELLED

```
public static final java.lang.String ST_USERCANCELLED
```

Status constant that indicates that the used does not want to go trhough with the download operation. The event may occur after the analyses of the Download Descriptor, or instead of the sedning of the Installation Notification (i.e. the user cancelled the download while th retrieval/installation of the media object was in process)

Constructors

OTA()

```
public OTA()
```

Methods

createDescriptor(`java.lang.String` uri)

```
public Descriptor₄₅ createDescriptor(java.lang.String uri)
            throws OTAException
```

Returns a decriptor information of an application.

Parameters:

uri - uri to get the descriptor from, usually returned by `discover()`₆₅ method. Upon encountering an error in the descriptor, this method will throw an `OTAException`₆₇ exception, and if url to notify server of problems is known, report the corresponding error to the server. It will not though send the success status, the application is responsible for doing that, since it may encounter a descriptor error itself.

Throws:

`OTAException`₆₇ - if there are troubles downloading or parsing the descriptor.

discover(`java.lang.String` url)

```
public java.util.Hashtable discover(java.lang.String url)
```

Discover applications at given URL. Downloads the page from the url and stores application name and descriptor in the hashtable, where urls are keys and application names are values.

Parameters:

url - url to discover from

Returns: list of discovered applications. If returned hashtable is empty, either no applications were discovered, or there was some sort of an error during the discovery.

download(`Descriptor`₄₅ des, `Destination`₅₇ store, `DLIndicator`₆₀ report)

```
public boolean download(Descriptor₄₅ des, Destination₅₇ store, DLIndicator₆₀ report)
            throws OTAException
```

Download the data identified by the associated Descriptor.

Parameters:

des - the Descriptor object identifying the download, which is provided by the `createDescriptor()`₆₅ method.

store - implementation instance of Destination interface to be used to store downloaded bits.

report - class implementing `DLIndicator`₆₀ interface. If null, reporting is not done.

Returns: false if download was cancelled. See `DLIndicator`₆₀ to learn how to cancel a download:

Throws:

`OTAException`₆₇ - if an error occured during download.

getSchema()

```
public abstract java.lang.String getSchema()
```

Returns schema implemented by this instance.

sendNotify(`java.lang.String uri`, `java.lang.String statusCode`, `java.lang.String statusMsg`)

```
public boolean sendNotify(java.lang.String uri, java.lang.String statusCode, java.lang.
          String statusMsg)
```

Send installation notification to the server, either reporting a success or a failure. Note, that some of the errors will already be reported by `download()` $_{65}$ method. So, if an exception was thrown from `download()` $_{65}$ method, you do not need to report another failure.

Parameters:

`uri` - **uri to send the status to. Usually this uri is taken from the descriptor information provided by the `createDescriptor()` $_{65}$ method.**

`statusCode` - **Status code to return. Since codes may be different in different OTA implementations, the special constants defined by this class should be used for that. The list of constants starts from `ST_SUCCESS` $_{64}$. Note, that not all of the status may be supported by all of the protocols. If the protocol does not support a certain constants, some default will be used.**

com.sun.appmanager.ota
OTAException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--com.sun.appmanager.ota.OTAException
```

All Implemented Interfaces: java.io.Serializable

Declaration

public class **OTAException** extends java.lang.Exception

Member Summary
Constructors
OTAException$_{67}$(java.lang.String reason)

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)
Methods inherited from class `Throwable`
fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()

Constructors

OTAException(`java.lang.String` reason)

```
    public OTAException(java.lang.String reason)
```

com.sun.appmanager.ota
OTAFactory

```
java.lang.Object
  |
  +--com.sun.appmanager.ota.OTAFactory
```

Declaration
public abstract class **OTAFactory**

Description
Abstract class providing a factory-like source for OTA download agents.

Member Summary	
Constructors	
	OTAFactory$_{68}$()
Methods	
OTA$_{62}$	getImpl$_{68}$(java.lang.String impl)

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

OTAFactory()

public **OTAFactory**()

Methods

getImpl(`java.lang.String` impl)

public OTA$_{62}$ **getImpl**(java.lang.String impl)

Return implementation instance.

Parameters:

`impl` **- protocol name, which is used to attempt to find an impl class for the protocol (e.g., "MIDP" or "OMA"). This argument can also denote a fully qualified class name to be instantiated.**

Returns: **OTA implementation class, or `null` if no implementation found for this protocol. If class name is specified, the class should have argumentless public constructor.** com.sun.

appmanager.ota
OTANotSupportedException

```
java.lang.Object
  |
  +--java.lang.Throwable
       |
       +--java.lang.Exception
            |
            +--java.lang.RuntimeException
                 |
                 +--com.sun.appmanager.ota.OTANotSupportedException
```

All Implemented Interfaces: java.io.Serializable

Declaration

public class **OTANotSupportedException** extends java.lang.RuntimeException

Member Summary
Constructors
OTANotSupportedException$_{70}$()
OTANotSupportedException$_{70}$(java.lang.String reason)

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)
Methods inherited from class `Throwable`
fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()

Constructors

OTANotSupportedException()

```
public OTANotSupportedException()
```

OTANotSupportedException(java.lang.String reason)

```
public OTANotSupportedException(java.lang.String reason)
```

com.sun.appmanager.ota
SyntaxException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--com.sun.appmanager.ota.SyntaxException
```

All Implemented Interfaces: `java.io.Serializable`

Declaration
`public class` **`SyntaxException`** `extends java.lang.Exception`

Description
Indicates an error encountered while parsing a download descriptor.

Member Summary
Constructors
SyntaxException$_{71}$()
SyntaxException$_{72}$(java.lang.String s)

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)
Methods inherited from class `Throwable`
fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()

Constructors

SyntaxException()

```
public SyntaxException()
```

SyntaxException(`java.lang.String` s)

```
public SyntaxException(java.lang.String s)
```

Package
com.sun.appmanager.preferences

Description

This is the layer handling persistent user and system preferences. These are string based key-value pairs, and are to be handled separately for users and the system. Persistence must be built into the system, and is expressed via the `save` methods below.

A `com.sun.appmanager.users.UserManager` instance can decide to manipulate user and system preferences when users are added or deleted. Also, a user or an administrator might set preferences via a screen in the presentation manager.

Class Summary	
Interfaces	
Preferences₇₄	An interface that defines access to preferences.

com.sun.appmanager.preferences
Preferences

Declaration
```
public interface Preferences
```

Description
An interface that defines access to preferences.

Preferences are string based key-value pairs.

There are two kinds of preferences: *user* preferences, and *system* preferences. User preferences can be thought of as multiple string-indexed preference *namespaces*. Preferences for one user are completely disjoint from preferences for another user. Overlap of a preference name for different user names is therefore not a conflict. System preferences on the other hand are *global*; ie they apply to the entire system regardless of the user namespaces.

An implementation of Preferences should allow for persistence of user and system preferences. The `save*` methods below handle persistence explicitly.

Member Summary	
Methods	
void	deleteUserPreferences₇₄(java.lang.String userName)
java.lang.String[]	getAllSystemPreferenceNames₇₄()
java.lang.String[]	getAllUserNames₇₅()
java.lang.String[]	getAllUserPreferenceNames₇₅(java.lang.String user)
java.lang.String	getSystemPreference₇₅(java.lang.String prefName)
java.lang.String	getUserPreference₇₅(java.lang.String userName, java.lang. String prefName)
void	saveAll₇₅()
void	saveSystemPreferences₇₅()
void	saveUserPreferences₇₅(java.lang.String userName)
void	setSystemPreference₇₅(java.lang.String prefName, java.lang. String prefValue)
void	setUserPreference₇₅(java.lang.String userName, java.lang. String prefName, java.lang.String prefValue)

Methods

deleteUserPreferences(java.lang.String userName)

```
public void deleteUserPreferences(java.lang.String userName)
```

Save all preferences for user `userName`

getAllSystemPreferenceNames()

```
public java.lang.String[] getAllSystemPreferenceNames()
```

Get names of all system preferences

getAllUserNames()

```
public java.lang.String[] getAllUserNames()
```

Get names of all users

getAllUserPreferenceNames(`java.lang.String` user)

```
public java.lang.String[] getAllUserPreferenceNames(java.lang.String user)
```

Get names of all user preferences corresponding to 'user'

getSystemPreference(`java.lang.String` prefName)

```
public java.lang.String getSystemPreference(java.lang.String prefName)
```

Get system preference `prefName`

getUserPreference(`java.lang.String` userName, `java.lang.String` prefName)

```
public java.lang.String getUserPreference(java.lang.String userName, java.lang.String
            prefName)
```

Get preference `prefName` for user `userName`.

saveAll()

```
public void saveAll()
```

Make all user preferences and system preferences persistent

saveSystemPreferences()

```
public void saveSystemPreferences()
```

Save all system preferences

saveUserPreferences(`java.lang.String` userName)

```
public void saveUserPreferences(java.lang.String userName)
```

Save preferences for user `userName`

setSystemPreference(`java.lang.String` prefName, `java.lang.String` prefValue)

```
public void setSystemPreference(java.lang.String prefName, java.lang.String prefValue)
```

Set system preference `prefName` to `prefValue`

setUserPreference(`java.lang.String` userName, `java.lang.String` prefName, `java.lang.String` prefValue)

```
public void setUserPreference(java.lang.String userName, java.lang.String prefName,
            java.lang.String prefValue)
```

Set preference `prefName` to `prefValue` for user `userName`.

Package
com.sun.appmanager.presentation

Description

Provides the classes and interfaces for a presentation layer. The presentation layer is free to implement any kind of user experience, accessing AppManager components where necessary via the AppManager$_{30}$ class. As such, its interface is very simple; it is probably the biggest *client* of the other modules of the system.

Class Summary	
Interfaces	
Presentation$_{78}$	This is a rudimentary interface to link a presentation layer with the startup code.
Classes	
PresentationMode$_{79}$	

com.sun.appmanager.presentation
Presentation

All Known Implementing Classes: PresentationMode$_{79}$

Declaration
```
public interface Presentation
```

Description
This is a rudimentary interface to link a presentation layer with the startup code. An implementation of AppManager$_{30}$ gets started, and in turn invokes methods in Presentation to get the presentation layer kick-started.

Member Summary	
Methods	
void	initialize$_{78}$()
void	loadApps$_{78}$()
void	runStartupApps$_{78}$()
void	startAppManager$_{78}$()

Methods

initialize()

```
public void initialize()
```

Initialize presentation mode in preparation for display to the user.

loadApps()

```
public void loadApps()
```

Load any application descriptors that might be displayed to the user

runStartupApps()

```
public void runStartupApps()
```

Run any applications that the presentation layer feels is appropriate to always display to the user.

startAppManager()

```
public void startAppManager()
```

Show the presentation mode and start interacting with the user.

com.sun.appmanager.presentation
PresentationMode

```
java.lang.Object
  |
  +--com.sun.appmanager.presentation.PresentationMode
```

All Implemented Interfaces: Presentation$_{78}$

Declaration

public abstract class **PresentationMode** implements Presentation$_{78}$

Member Summary	
Constructors	
	PresentationMode$_{79}$()
Methods	
void	addApp$_{80}$(AppModule$_{20}$ module)
static PresentationMode$_{79}$	createPresentation$_{80}$()
static java.lang. String	getPresentationClassName$_{80}$()
void	runStartupApp$_{80}$(AppModule$_{20}$ module)

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)
Methods inherited from interface Presentation$_{78}$
initialize()$_{78}$, loadApps()$_{78}$, runStartupApps()$_{78}$, startAppManager()$_{78}$

Constructors

PresentationMode()

    public **PresentationMode**()

Methods

addApp(**AppModule**$_{20}$ module)

```
public void addApp(AppModule20 module)
```

Add an application or menu represented by an AppModule to this presentation mode. It is highly recommended that this method be used within the implemented loadApps() method of the Presentation interface.

Parameters:

module - the module to be added

createPresentation()

```
public static PresentationMode79 createPresentation()
```

getPresentationClassName()

```
public static java.lang.String getPresentationClassName()
```

Return the presentation mode's implementation class name of the Presentation interface. The returned string will contain of the following format represent by this example of a presentation mode called "AwtPDA": com.sun.appmanager.impl.presentation.AwtPDAPresentationMode

Returns: String: runStartupApp(AppModule$_{20}$ module)

```
public void runStartupApp(AppModule20 module)
```

Launch the application upon startup of the application manager. It is highly recommended that this method be used within the implemented runStartupApps() method of the Presentation interface.

Parameters:

module - AppModule

Package
com.sun.appmanager.store

Description

An abstraction of persistent storage. `PersistentStore`$_{82}$ allows use of abstract relative paths, and converts these to absolute paths for access via java.io file API's.

Class Summary	
Interfaces	
`PersistentStore`$_{82}$	Generic interface to be used to specify a persistent storage implementation.

com.sun.appmanager.store
PersistentStore

Declaration
```
public interface PersistentStore
```

Description
Generic interface to be used to specify a persistent storage implementation. The current version allows the construction of full file paths out of relative storage paths.

Member Summary	
Methods	
java.lang.String	absolutePathOf$_{82}$(java.lang.String storePath)

Methods

absolutePathOf(`java.lang.String` storePath)

```
public java.lang.String absolutePathOf(java.lang.String storePath)
```

Parameters:

`storePath` - relative path of a file

Returns: the absolute file path in this `PersistentStore` implementation.

Constant Field Values

Contents

- com.sun.*

com.sun.*$_{13}$

com.sun.appmanager.appmodel.AppModelController$_{14}$		
static final int	XLET_APP_MODEL$_{14}$	1

com.sun.appmanager.appmodel.XletAppModelController$_{16}$		
static final int	XLET_ACTIVE$_{17}$	3
static final int	XLET_DESTROYED$_{17}$	4
static final int	XLET_LOADED$_{17}$	1
static final int	XLET_PAUSED$_{17}$	2
static final int	XLET_UNKNOWN$_{17}$	0

com.sun.appmanager.mtask.Client$_{34}$		
static final java.lang.String	DEFAULT_SERVER_NAME$_{35}$	"localhost"
static final int	DEFAULT_SERVER_PORT$_{35}$	7777

com.sun.appmanager.mtask.TaskListener$_{38}$		
static final int	CDCAMS_TASK_KILLED$_{38}$	1

com.sun.appmanager.ota.OTA[62]		
static final java.lang.String	ST_ATTRIBUTEMISMATCH[63]	"905"
static final java.lang.String	ST_INSUFFICIENTMEMORY[63]	"901"
static final java.lang.String	ST_INVALIDDDVERSION[63]	"951"
static final java.lang.String	ST_INVALIDDESCRIPTOR[63]	"906"
static final java.lang.String	ST_LOSSOFSERVICE[64]	"903"
static final java.lang.String	ST_SIZEMISMATCH[64]	"904"
static final java.lang.String	ST_SUCCESS[64]	"900"
static final java.lang.String	ST_USERCANCELLED[64]	"902"

Contents

This part contains the appendices:

- Debugging with gdb
- C Stack Checking

Debugging with `gdb`

This chapter contains tips for debugging the CDC-HI Java virtual machine with `gdb`.
It covers these topics:

- Setup Procedures
  - Signal Handlers
  - `gdb` and GC Safety
  - Turning on Trace Output
- High-Level Dumpers
  - `CVMdumpObject`
  - `CVMdumpClassBlock`
  - `CVMdumpString`
  - `CVMdumpObjectReferences`
  - `CVMdumpClassReferences`
- Low-Level Dumpers
  - Using `CVMconsolePrintf()`
  - Displaying the PC Offset
  - Dumping the Java Stack
  - Displaying Opcode Information
  - Dumping the Java Heap
  - Dumping Object Information
  - Dumping Loaded Classes
  - Dumping Threads
- Conversion Procedures
  - The `CVMExecEnv` Structure
  - Converting Between `CVMExecEnv*` and `JNIEnv*`
  - Converting from JNI Types to Internal VM Types
  - Converting from `java.lang.Class` to `CVMClassBlock*`
- Other Procedures
  - Debugging Crashes on Linux
  - Debugging Compiled Methods

This chapter describes `gdb` techniques that are specific to CDC runtime
development. See the `gdb` reference manual listed in the Preface for more
information about `gdb`.

A.1 Setup Procedures

A.1.1 Signal Handlers

There are a number of signals that are raised by the CDC-HI virtual machine that `gdb` must be made aware of so it can pass them on. You need to execute the following `handle` commands to avoid having `gdb` stop execution unnecessarily:

```
handle SIGUSR1 nostop noprint pass

handle SIGUSR2 nostop noprint pass

handle SIGSTOP nostop noprint pass
```

For `CVM_JIT=true` builds, the target port usually chooses to use trap-based null checks by defining `CVMJIT_TRAP_BASED_NULL_CHECKS` in `src/<OS>-<CPU>/javavm/runtime/jit/jit_arch.h`. In this case you will also need to execute:

```
handle SIGSEGV nostop noprint pass
```

and set a breakpoint in your implementation of `handleSegv()`. `handleSegv()` is usually located in `src/<OS>-<CPU>/javavm/runtime/jit/jit_arch.c`. `handleSegv()` needs to handle three cases of `SIGSEGV`. The first and second cases are for a `SIGSEGV`s that occur in JIT compiled and supporting code respectively. These cases are interpreted as `NullPointerExceptions` and should be ignored by `gdb`. (This is only necessary for applications that cause `NullPointerException`.) The third case for `SIGSEGV` represents a crash in the VM. This case is where the breakpoint should be set to inform you of a crash when it occurs. If you reach this breakpoint, then use

```
handle SIGSEGV stop

continue
```

to reproduce the crash at the instruction where it occurred.

Note – Because `SIGSEGV` is used in JIT compiled code to represent `NullPointerExceptions`, crashes that occur in JIT compiled code that has not been fully debugged may be mis-interpreted as a `NullPointerException` and may not show up as a crash at all. If you need to eliminate this possibility for debugging purposes, you can undefine `CVMJIT_TRAP_BASED_NULL_CHECKS` in `src/<OS>-<CPU>/javavm/runtime/jit/jit_arch.h`. This will cause `SIGSEGV` to not be used for null checks, and any `SIGSEGV` that you get will actually be due to a crash. This can also be done to avoid handling `SIGSEGV` as described earlier.

A.1.2   gdb and GC Safety

There are some things you can't do in the debugger while GC unsafe, such as calling `CVMdumpThread()`. Examine `ee->tcstate.isConsistent`. It must be 1. If it is not, then you can try switching to a thread that is. If this isn't possible, you can always try the following:

```
(gdb) set var ee->tcstate.isConsistent = 1
(gdb) call CVMdumpAllThreads()
```

However, there is a very small risk of a deadlock or a crash if you do this. Don't forget to set the `isConsistent` flag back to 0 if you wish to continue execution. You are always GC safe when executing in a JNI method and are usually GC safe when executing in a JNI API or any class loading or unloading related code. You are usually always GC unsafe when executing in the interpreter loop (`CVMgcUnsafeExecuteJavaMethod()`) or in dynamically compiled code.

A.1.3   Turning on Trace Output

If the CDC-HI virtual machine is built with `CVM_TRACE=true`, then the support for debug tracing will be compiled in. There are three ways to turn on trace flags. By default, `CVM_TRACE=true` if `CVM_DEBUG=true`.

1. Use the `-Xtrace` command line argument:

```
cvm -Xtrace:0xc40000 -Djava.class.path=../testclasses HelloWorld
```

2. Turn the flags on or off manually in `gdb`:

```
(gdb) set var CVMglobals.debugFlags = 0xc40000
```

3. Turn flags on and off from Java source code:

You must first import `sun.misc.CVM`. You can then use the following APIs:

```
/*
 * Methods for checking, setting, and clearing the state of debug
 * flags.  All of the following methods return the previous state of
 * the flags.
 *
 * You can pass in more than one flag at a time to any of the methods.
 */
public native static int checkDebugFlags(int flags);
public native static int setDebugFlags(int flags);
public native static int clearDebugFlags(int flags);
```

```
public native static int restoreDebugFlags(int flags, int oldvalue);
```

See `src/share/javavm/test/Test.java` for an example on using these APIs.

The supported trace flags can be found in `src/share/classes/sun/misc/CVM.java` or see the companion document *CDC Runtime Guide* for a documented list. To turn on more than one flag at the same time, use a logical OR of their values:

TABLE A-1 Trace Flag Values

Flag	Value
DEBUGFLAG_TRACE_OPCODE	0x00000001
DEBUGFLAG_TRACE_METHOD	0x00000002
DEBUGFLAG_TRACE_STATUS	0x00000004
DEBUGFLAG_TRACE_FASTLOCK	0x00000008
DEBUGFLAG_TRACE_DETLOCK	0x00000010
DEBUGFLAG_TRACE_MUTEX	0x00000020
DEBUGFLAG_TRACE_CS	0x00000040
DEBUGFLAG_TRACE_GCSTARTSTOP	0x00000080
DEBUGFLAG_TRACE_GCSCAN	0x00000100
DEBUGFLAG_TRACE_GCSCANOBJ	0x00000200
DEBUGFLAG_TRACE_GCALLOC	0x00000400
DEBUGFLAG_TRACE_GCCOLLECT	0x00000800
DEBUGFLAG_TRACE_GCSAFETY	0x00001000
DEBUGFLAG_TRACE_CLINIT	0x00002000
DEBUGFLAG_TRACE_EXCEPTIONS	0x00004000
DEBUGFLAG_TRACE_MISC	0x00008000
DEBUGFLAG_TRACE_BARRIERS	0x00010000
DEBUGFLAG_TRACE_STACKMAPS	0x00020000
DEBUGFLAG_TRACE_CLASSLOADING	0x00040000
DEBUGFLAG_TRACE_CLASSLOOKUP	0x00080000
DEBUGFLAG_TRACE_TYPEID	0x00100000
DEBUGFLAG_TRACE_VERIFIER	0x00200000

TABLE A-1   Trace Flag Values

Flag	Value
DEBUGFLAG_TRACE_WEAKREFS	0x00400000
DEBUGFLAG_TRACE_CLASSUNLOAD	0x00800000
DEBUGFLAG_TRACE_CLASSLINK	0x01000000

TABLE A-2 lists the debug flag trace options most commonly used. The remainder of the flags are less commonly used.

TABLE A-2   Debug Flag Trace Options

Option	Description
OPCODE	Traces each opcode being executed.
METHOD	Traces each method as it is entered, exited, and returned to.
GCSTARTSTOP	Provides information each time GC starts and completes.
EXCEPTIONS	Does a full backtrace of the Java thread whenever an exception is thrown, and also provide information when an exception is caught.
CLASSLOADING	Provides information about classes being loaded.
CLASSLOOKUP	Provides information about class lookups (mostly loader cache related information).
CLASSUNLOAD	Provides information each time a class is unloaded.
CLASSLINK	Provides information when a class is linked.

There are also a number of dynamic compiler related trace flags that are described in the companion document *CDC Runtime Guide*. These trace flags can be set with the following techniques:

- `-Xjit=trace=<option>` command-line options
- `CVM.setJITDebugFlags()` method found in `CVM.java`
- `CVMglobals.debugJITFlags` command in `gdb`

A.2   High-Level Dumpers

The following dumper utilities display runtime information in a more useful format. These utilities are only available when the VM is built with `CVM_DEBUG=true` and can be used at anytime except during garbage collection.

Before using these utilities, you should disable GC:

```
(gdb) call CVMgcDisableGC()
$2 = 1
```

Only the first invocation of `CVMgcDisableGC` will have any effect.

```
(gdb) call CVMgcDisableGC()
GC is already disabled!  No need to disable.
$3 = 1
```

When you're done, you can re-enable GC with `CVMgcEnableGC`. For example:

```
(gdb) call CVMgcEnableGC()
$4 = 1
```

Only the first invocation of `CVMgcEnableGC` will have any effect.

A.2.1   CVMdumpObject

`CVMdumpObject` dumps the contents of an object using its direct object pointer. For example:

```
(gdb) call CVMdumpObject(directObj)
Object 0x61034c: instance of class java.lang.ThreadGroup
  classblock = 0x40ddfc
  size = 48
  fields[10] = {
    [000b] 0x0 : groups:[Ljava/lang/ThreadGroup;
    [000a] 0 : ngroups:I
    [0009] 0x0 : threads:[Ljava/lang/Thread;
    [0008] 0 : nthreads:I
    [0007] 0 : vmAllowSuspension:Z
    [0006] 0 : daemon:Z
    [0005] 0 : destroyed:Z
    [0004] 10 : maxPriority:I
    [0003] 0x51a2ec : name:Ljava/lang/String;
    [0002] 0x0 : parent:Ljava/lang/ThreadGroup;
}
```

NOTE: `CVMdumpObject` operates on the value of a direct object pointer (`CVMObject` pointer or `CVMObjectICell`). An `CVMObjectICell` pointer will need to be dereferenced before `CVMdumpObject` can be used. If you accidentally typed in a wrong object pointer value, `CVMdumpObject` will fail safely and tell you that you do not have a valid object. For example:

```
(gdb) call CVMdumpObject(235)
Address 0xeb is not a valid object
```

A.2.2   CVMdumpClassBlock

`CVMdumpClassBlock` dumps the contents of a classblock. For example:

```
(gdb) call CVMdumpClassBlock(0x40ddfc)
Classblock 0x40ddfc: class java.lang.ThreadGroup
  class object = 0x549404
  instance size = 48
  superclass = 0x3ee05c : java.lang.Object
  classloader ref= 0x0, obj= 0x0 :
  protectionDomain = 0x0
  instanceSize = 48
  static fields [0] = { NONE }
```

NOTE: The classblock is not the class object (`CVMClassBlock *`). The pointer for class object is the first field given in the classblock dump. You can use `CVMdumpObject` to dump the class object. For example:

```
(gdb) call CVMdumpObject(0x549404)
Object 0x549404: instance of class java.lang.Class
classblock = 0x3edd70
size = 16
fields[2] = {
  [0003] 0x0 : loader:Ljava/lang/ClassLoader;
  [0002] 4251132 : classBlockPointer:I
}
```

NOTE: The classblock pointer is also contained in the class object. If you type in a wrong classblock pointer value, `CVMdumpClassBlock` will fail gracefully and tell you so. For example:

```
(gdb) call CVMdumpClassBlock(0x40ddf0)
Address 0x40ddf0 is not a valid Classblock
```

NOTE: A classblock pointer for an object can be obtained by either calling `CVMdumpObject` or by masking the lower two bits off of the first word of the object.

A.2.3   `CVMdumpString`

`CVMdumpString` dumps the contents of a `java.lang.String` object as a C string. Note that this dumper does not go through any character decoder. It simply truncates the Java `chars` and dumps the string as ASCII. This should be adequate for most debugging purposes that don't concern localization. For example:

```
(gdb) call CVMdumpString(0x51a2ec)
String 0x51a2ec: length= 6
value= "system"
```

NOTE: You can also call `CVMdumpObject(directObj)` to dump the string, but its contents will show as a char array instead of the above nicely formatted string.

If you accidentally specified a wrong `String` object pointer, `CVMdumpString` will fail gracefully and tell you so. For example:

```
(gdb) call CVMdumpString(254)
```

Address 0xfe is not a valid object.

A.2.4   `CVMdumpObjectReferences`

`CVMdumpObjectReferences` dumps all references to a specific object, which is useful for discovering what is keeping an object alive.

For example:

```
(gdb) call CVMdumpObjectReferences(directObj)
List of references to object 0x61034c (java.lang.ThreadGroup):
  Ref 0x525c30 type: PRELOADER STATICS ROOT
  Ref 0x5fd510 type: JAVA FRAME ROOT ee 0x504ac0 frame 0
```

If you accidentally specified a wrong object pointer value, the dumper will fail gracefully and tell you so. For example:

```
(gdb) call CVMdumpObjectReferences(555)
```

Address `0x22b` is not a valid object.

A.2.5   CVMdumpClassReferences

CVMdumpClassReferences dumps all references to a class by name. For example:

```
(gdb) call CVMdumpClassReferences("java/lang/ThreadGroup")
Addr: 0x61034c Size: 48      Class: java.lang.ThreadGroup
List of references to object 0x61034c (java.lang.ThreadGroup):
  Ref 0x525c30 type: PRELOADER STATICS ROOT
  Ref 0x5fd510 type: JAVA FRAME ROOT ee 0x504ac0 frame 0
Addr: 0x61037c Size: 48      Class: java.lang.ThreadGroup
List of references to object 0x61037c (java.lang.ThreadGroup):
  Ref 0x5fd50c type: JAVA FRAME ROOT ee 0x504ac0 frame 0
  Ref 0x5fd508 type: JAVA FRAME ROOT ee 0x504ac0 frame 1
```

NOTE: The classname uses '/' as a separator instead of '.'. The dumper will not recognize '.' as a separator. For example:

```
(gdb) call CVMdumpClassReferences("java.lang.ThreadGroup")
  Class java.lang.ThreadGroup is NOT loaded
```

The error message is misleading in this case (an unfortunate side effect of the current implementation).

A.3   Low-Level Dumpers

A.3.1   Using CVMconsolePrintf()

CVMconsolePrintf() provides information about classes, methods, fields, objects, and Java stack frames. CVMconsolePrintf() supports six special conversion characters in addition to those normally supported by printf(), plus the meaning of three of the characters can be modified with the '!' character. The conversion characters include:

```
%C and %!C - prints a class name
%M and %!M - prints a method name
%F and %!F - prints a field name
%O and %I  - prints out an object's class name and hash
%P         - prints out java stack frame information
```

The argument type required for each of the above conversion characters is as follows:

```
%C  - CVMClassBlock*
%M  - CVMMethodBlock*
%F  - CVMFieldBlock*
%!C - CVMClassTypeID
%!M - CVMMethodTypeID
%!F - CVMFieldTypeID
%O  - CVMObject*
%I  - CVMObjectICell*
%P  - CVMInterpreterFrame* (only supported in CVM_DEBUG=true builds)
```

You must be GC safe when using `%I`. See Section A.1.2, "gdb and GC Safety" on page A-3. `%O` will not print the object hash if it has not been calculated yet.

Here are some examples:

```
(gdb) call CVMconsolePrintf("%C.%M\n", cb, mb)
java.lang.Class.runStaticInitializers()V


(gdb) call CVMconsolePrintf("%O\n", directObj)
java.lang.Class@0


(gdb) call CVMconsolePrintf("%P\n", frame)
java.lang.Class.runStaticInitializers()V(Class.java:1446)
```

A.3.2   Displaying the PC Offset

Given a bytecode address (such as the `pc` local variable) you can find the offset from the start of the method. You will also need the `CVMMethodBlock*`, which can be retrieved from the frame of the method in order to do this:

```
(gdb) p *(CVMInterpreterFrame*)frame
$61 = {
  frameX = {
    prevX = 0x387950,
    scanner = 0xaf70c <CVMjavaFrameScanner>,
    topOfStack = 0x38799c,
    mb = 0x2bbb5c
  },
```

```
    pcX = 0x2bb6fb "ß\013\003 ",
    cpX = 0x2bbc58,
    localsX = 0x387970
}
(gdb) p $61->pcX - (CVMUint8*)($61->frameX.mb->immutX.codeX.jmd+1)
$62 = 7
```

You can define a macro `pcOffset(frame)` to print out the current offset of the pc from the start of the method:

```
define pcOffset
    p ((CVMInterpreterFrame*)$arg0)->pcX -
(CVMUint8*)(((CVMInterpreterFrame*)$arg0)->frameX.mb-
>immutX.codeX.jmd+1)
end
```

Note – This technique is only applicable if the frame is not a compiled frame. See Section A.3.4, "Displaying Opcode Information" on page A-12.

A.3.3 Dumping the Java Stack

There are two functions that you can call from `gdb` to dump out Java stack information. They are only included if you build with CVM_DEBUG_DUMPSTACK=true, which is the default if you build with CVM_DEBUG=true. If you also want source file and line number information included in the stack dump, then you need to also build with CVM_DEBUG_CLASSINFO=true and CVM_JAVAC_DEBUG=true, both of which also default to true if CVM_DEBUG=true.

```
extern void

CVMdumpStack(CVMStack* s, CVMBool verbose, CVMBool includeData,
CVMInt32 frameLimit);

extern CVMStackChunk*

CVMdumpFrame(CVMFrame* frame, CVMStackChunk* startChunk, CVMBool
verbose, CVMBool includeData);

(gdb) call CVMdumpStack(&ee->interpreterStack,0,0,0)

  Java Frame Test.testSunMiscGC()V(Test.java:158)

  Java Frame Test.main([Ljava/lang/String;)V(Test.java:123)

  Transition Frame Test.main([Ljava/lang/String;)V(Transition
Method)

  Free List Frame (JNI Local Frame)
```

If you pass 1 for `verbose`, you will see more details for each frame. If you also pass 1 for `includeData`, then the stack contents for each frame are also displayed. Alternatively, you can use the special combination of `verbose==0` and `includeData==1` to get the minimal information for each frame plus the arguments needed to call `CVMdumpFrame()` for more information on each of the respective frames. For example:

```
(gdb) call CVMdumpStack(&ee->interpreterStack,0,1,0)
  Java Frame Test.testSunMiscGC()V(Test.java:158)
call CVMdumpFrame(0x387878, 0x3877d8, 1, 1)
  Java Frame Test.main([Ljava/lang/String;)V(Test.java:123)
call CVMdumpFrame(0x38784c, 0x3877d8, 1, 1)
  Transition Frame Test.main([Ljava/lang/String;)V(Transition
Method)
call CVMdumpFrame(0x38781c, 0x3877d8, 1, 1)
  Free List Frame (JNI Local Frame)
call CVMdumpFrame(0x3877e4, 0x3877d8, 1, 1)
```

The advantage of this output is that arguments needed to call `CVMdumpFrame()` are automatically included for you, so you don't need to look at the verbose output of `CVMdumpStack()` to figure out which arguments to pass to `CVMdumpFrame()`, and you don't need to deal with a stack dump that includes the stack data for every frame. For example:

```
(gdb) call CVMdumpFrame(0x387878, 0x3877d8, 1, 1)
        Frame:     0x387878
        prev:      0x38784c
        Scanner:   0xaf70c
        Tos:       0x387898
        Type:      Java Frame
        Name:      Test.testSunMiscGC()V(Test.java:158)
        NextPC:    0x8be023
        special:   0
        Contents:
            Chunk 0x3877d8 (0x3877e4-0x3887e4)
```

A.3.4 Displaying Opcode Information

There are three ways to locate the current PC for a frame:

1. Use `CVMdumpStack()` or `CVMdumpFrame()` and pass `verbose==1`. The address of the current program counter (pc) will be included in the output in the `NextPC` field.

```
(gdb) call CVMdumpFrame(0x387878, 0x3877d8, 1, 1)
        Frame:    0x387878
        prev:     0x38784c
        Scanner:  0xaf70c
        Tos:      0x387898
        Type:     Java Frame
        Name:     Test.testSunMiscGC()V(Test.java:158)
        NextPC:   0x8be023
        special:  0
        Contents:
            Chunk 0x3877d8 (0x3877e4-0x3887e4)
```

2. Locate the `CVMFrame*` for the frame and display its contents. The current pc should be in the `pcX` field.

```
(gdb) p *(CVMInterpreterFrame*) frame
$46 = {
  frameX = {
    prevX = 0x38797c,
    scanner = 0xaf70c <CVMjavaFrameScanner>,
    topOfStack = 0x38798c,
    mb = 0x2bbb6c
  },
  pcX = 0x3879c8 "",
  cpX = 0x2bbc58,
  localsX = 0x38799c
}
```

Note that:

- These techniques work with Java frames, but not JNI frames, which don't have a pc.
- The `scanner` field must be set to `<CVMjavaFrameScanner>`.
- `frame->pcX` may not be correct for the topmost frame on the stack, since it may be cached in a local variable. Use the `pc` local variable in `CVMgcUnsafeExecuteJavaMethod()` instead.

3. Go to the `CVMgcUnsafeExecuteJavaMethod()` C frame and display the `pc` variable:

```
(gdb) p pc
$47 = (CVMUint8 *) 0x2bb74e " "
```

Once you have a pc, you can display the opcode at the pc:

```
(gdb) p (CVMOpcode)*(CVMUint8*)0x2bb74e
$48 = opc_return
```

For `CVM_JIT=true` builds, it does not make sense to display opcode information when the frame is a compiled frame (as opposed to an interpreted Java frame). This is because the code being executed is not the original bytecodes but a compiled version of it.

Regardless, when dumping stack frames, you need to be able to distinguish between compiled and interpreted frames. For `CVM_JIT=true` builds, the low bit of `prevX` for stack frames will normally be set. If the low bit is not set, then the scanner field contains unreliable information. If the low bit is not set, the frame is a compiled frame even if the scanner field says `<CVMjavaFrameScanner>`. If the low bit is set, for compiled frames, the scanner field will indicate `<CVMcompiledFrameScanner>`.

Like JNI native code frames, you cannot get program counter and opcode information from compiled frames.

A.3.5 Dumping the Java Heap

There are three functions that can be used for dumping out the contents of the Java heap. All of these functions can be useful in detecting leaks in the Java heap. They are only available if you build with `CVM_DEBUG=true`.

```
extern void CVMgcDumpHeapSimple()
extern void CVMgcDumpHeapStats()
extern void CVMgcDumpHeapVerbose()
```

`CVMgcDumpHeapSimple()` dumps the total number of objects in the heap. For example:

```
(gdb) call CVMgcDumpHeapSimple()
Counted 3702 objects
```

`CVMgcDumpHeapStats()` displays the number of instances (`NI`) allocated for each class and the total space (`TS`) in bytes that the instances occupy in the heap. For example:

```
(gdb) call CVMgcDumpHeapStats()
TS=89396    NI=986      CL=[C
TS=47172    NI=167      CL=[B
```

```
TS=14292      NI=397        CL=[Ljava.lang.Object;
TS=14020      NI=701        CL=java.lang.String
TS=9056       NI=126        CL=[I
TS=8096       NI=2          CL=[S
TS=3700       NI=185        CL=java.lang.Class
TS=3480       NI=174        CL=java.lang.StringBuffer
TS=2256       NI=94         CL=java.util.Hashtable$Entry
TS=2040       NI=18         CL=[Ljava.util.Hashtable$Entry;
TS=1400       NI=116        CL=[Ljava.lang.Class;
...
TS=8          NI=1          CL=java.util.Hashtable$EmptyEnumerator
TS=8          NI=1          CL=java.net.UnknownContentHandler
TS=8          NI=1          CL=java.security.Security$1
TS=8          NI=1          CL=java.util.Hashtable$EmptyIterator
```

CVMgcDumpHeapVerbose() dumps out the address and size of every object in the heap. For example:

```
(gdb) call CVMgcDumpHeapVerbose()
...
Addr: 0x3bd0a8 Size: 44  Class: CloneableObject
Addr: 0x3bd0d4 Size: 8   Class: java.lang.Object
Addr: 0x3bd0dc Size: 44  Class: CloneableObject
Addr: 0x3bd108 Size: 24  Class: [Ljava.lang.Object;
Addr: 0x3bd120 Size: 24  Class: [Ljava.lang.Object;
Addr: 0x3bd138 Size: 12  Class: NonCloneableObject
Addr: 0x3bd144 Size: 16 Class: java.lang.CloneNotSupportedException
...
```

The next section describes how you can dump the contents of these objects.

A.3.6   Dumping Object Information

Objects have an 8 byte header, followed by the contents of the object.

```
(gdb) x /4wx 0x3bd144
0x3bd144:    0x002ebffc  0x00000002  0x003bd1a0  0x003bd154
(gdb) p *(CVMObject*) 0x3bd144
$74 = {
```

```
      hdr = {
        clas = 0x2ebffc,
        various32 = 2
      },
      ...
    }
```

The `clas` field contains the `CVMClassBlock*` that the object is an instance of. The lower two bits of this field have special meaning and should be masked off if set before attempting to use it as a `CVMClassBlock*`.

```
(gdb) p ((CVMObject*)0x3bd144)->hdr.clas
$76 = (CVMClassBlock *) 0x2ebffc
(gdb) call CVMconsolePrintf("%C\n", $76)
java.lang.CloneNotSupportedException
```

Array objects have the same header as above, with the addition of a length field. Below is an array of length one.

```
(gdb) x /4wx 0x3bd7d8
0x3bd7d8:   0x008a8ce0   0x00000002   0x00000001   0x00000000
(gdb) p *((CVMArrayOfAnyType*)0x3bd7d8)
$77 = {
  hdr = {
    clas = 0x8a8ce0,
    various32 = 2
  },
  length = 1,
  ...
}
```

See the description of the high-level dumper in Section A.2.1, "CVMdumpObject" on page A-6.

A.3.7 Dumping Loaded Classes

You can dump the set of loaded classes using the `CVMclassTableDump()` function. You must first enable `CLASSLOOKUP` tracing or no output will be displayed. You must also compile with `CVM_TRACE=true` which is the default when `CVM_DEBUG=true`.

```
(gdb) set var CVMglobals.debugFlags = 0x80000
```

```
(gdb) call CVMclassTableDump(ee)
CT: 0x8d5db8: sun.misc.GC$1
CT: 0x8d53a8: sun.misc.GC$Daemon
CT: 0x8d4900: java.util.TreeMap$Entry
CT: 0x8d3f18: java.util.TreeMap$1
CT: 0x8d7b40: java.util.TreeMap
CT: 0x8d3190: java.util.TreeSet
CT: 0x8d0ed8: sun.misc.GC$LatencyRequest
CT: 0x8cfa18: sun.misc.GC$LatencyLock
CT: 0x8ce568: sun.misc.GC
```

The addresses listed are for the CVMClassBlock* of each class. You can use a
CVMClassBlock* address as follows to get more information for any individual
class:

```
(gdb) p *(CVMClassBlock*)0x8d5db8
$64 = {
  gcMapX = {
    map = 0,
    bigmap = 0x0
  },
  classNameX = 978,
  superclassX = {
    superclassCb = 0x244b2c,
    superclassTypeID = 2378540,
    mirandaMethodCountX = 0 '\000'
  },
  cpX = {
    constantpoolX = 0x8d5e20,
    arrayInfoX = 0x8d5e20
  },
  ...
}
```

You can also dump out the loader cache using CVMloaderCacheDump(). Once
again, CLASSLOOKUP tracing must be turned on and you must compile with
CVM_TRACE=true and CVM_DEBUG=true.

```
(gdb) call CVMloaderCacheDump(ee)
LC: #887 0x8a8fc0: <0x38a0a4,[LTest;>
```

```
LC: #890 0x8a8e00: <0x38a0a4,[LC;>
LC: #895 0x8ccb88: <0x38a0a4,[Lcvmtest.TypeidRefcountHelper;>
LC: #978 0x8d5db8: <0x0,sun.misc.GC$1>
LC: #979 0x8cfa18: <0x0,sun.misc.GC$LatencyLock>
LC: #980 0x8d53a8: <0x0,sun.misc.GC$Daemon>
LC: #981 0x8d3190: <0x0,java.util.TreeSet>
LC: #982 0x8d7b40: <0x0,java.util.TreeMap>
LC: #983 0x8d4900: <0x0,java.util.TreeMap$Entry>
LC: #984 0x8d3f18: <0x0,java.util.TreeMap$1>
LC: #1007 0x8a2cf0: <0x38a0a4,[[LTest;>
```

The first address given is the `CVMClassBlock*`. The second is the
`CVMObjectICell*` of the ClassLoader instance. `0x0` represents the NULL class
loader (a.k.a. the bootclasspath loader or bootstrap loader).

See the description of the high-level dumper in Section A.2.2,
"CVMdumpClassBlock" on page A-7.

A.3.8   Dumping Threads

Information about one or all of the Java threads can be dumped by using the
following functions, which are only available when the CDC-HI virtual machine is
built using `CVM_DEBUG=true`:

```
extern void
CVMdumpThread(JNIEnv* env)
extern void
CVMdumpAllThreads()
extern void
CVMprintThreadName(JNIEnv* env, CVMObjectICell* threadICell)
```

You must be GC safe when calling these functions. See Section A.1.2, "gdb and GC
Safety" on page A-3.

A.4 Conversion Procedures

A.4.1 The `CVMExecEnv` Structure

All Java threads in the CDC-HI virtual machine are represented by a `CVMExecEnv` structure. If you look at the C backtrace for almost any running thread, you will probably see a `CVMExecEnv*` passed as an argument to just about every function in the backtrace. You can use the `CVMExecEnv*` to locate information about the thread, such as the Java stack being used by the interpreter loop. For example, if your C backtrace is as follows:

```
#0   CVMgcUnsafeExecuteJavaMethod (ee=0x37d2c0, mb=0x267590,
     isStatic=0,
     isVirtual=0) at ../../src/share/javavm/runtime
     executejava.c:1636
#1   0xe4b2c in CVMjniInvoke (env=0x37d2e8, obj=0x387810,
     methodID=0x8bd124,
     pushArguments=0xe3638 <CVMjniPushArgumentsVararg>,
     args=0xffbef5b4,
     info=770, retValue=0x0) at ../../src/share/javavm/runtime/
     jni_impl.c:2412
#2   0xe727c in CVMjniCallStaticVoidMethod (env=0x37d2e8,
     clazz=0x387810,
     methodID=0x8bd124) at ../../src/share/javavm/runtime/
     jni_impl.c:2587
#3   0x19a644 in ansiJavaMain (argc=3, argv=0xffbef754)
     at ../../src/portlibs/ansi_c/ansi_java_md.c:223
#4   0x199cc4 in main (argc=3, argv=0xffbef754)
     at ../../src/solaris/bin/java_md.c:16
```

The ee argument passed to `CVMgcUnsafeExecuteJavaMethod()` can be used to dump the Java stack:

```
(gdb) call CVMdumpStack(&ee->interpreterStack,0,0,0)
```

You can also always get the `CVMExecEnv*` for the current thread by calling `CVMgetEE()`.

A.4.2 Converting Between `CVMExecEnv*` and `JNIEnv*`

Every `CVMExecEnv` has a corresponding `JNIEnv`. You can manually convert between the two, but usually you can avoid having to do this conversion by looking elsewhere on the C stack. For example, if you are in `CVMgcUnsafeExecuteJavaMethod()` and need a `JNIEnv*`, going up one or two C frames will usually put you in one of the JNI APIs, and you can get the `JNIEnv*` there.

Converting from `CVMExecEnv*` to `JNIEnv*`:

```
(gdb) p &(&(ee)->jniEnv)->vector
$86 = (JNIEnv *) 0x37d2e8
```

Converting from `JNIEnv*` to `CVMExecEnv*`:

```
(gdb) p (CVMExecEnv*)((char*)env - (CVMUint32)&(((CVMExecEnv*)0)-
>jniEnv)->vector)
$87 = (CVMExecEnv *) 0x37d2c0
```

Here are some conversion macros:

```
define ee2env
    p &(($arg0)->jniEnv)->vector
end

define env2ee
    ee2env((CVMExecEnv*)0)
    p (CVMExecEnv*)((char*)$arg0 - (CVMUint32)($))
end
```

A.4.3 Converting from JNI Types to Internal VM Types

The following are mappings of JNI types to internal VM types:

```
jclass == CVMClassICell* == java.lang.Class instance
jmethodID == CVMMethodBlock*
jfieldID == CVMFieldBlock*
jobject == CVMObject*
```

You can pass a variable of type `jmethodID` as the `%M` argument to `CVMconsolePrintf()`. Likewise for `jfieldID` and `%F`. Even though these types have "ID" in their names, they are not the same as `CVMMethodTypeID` and `CVMFieldTypeID`.

To print the type that a `jclass` represents, see the next section on converting from a `java.lang.Class` to `CVMClassBlock*`.

A.4.4 Converting from `java.lang.Class` to `CVMClassBlock*`

All `java.lang.Class` instances contain a pointer to the `CVMClassBlock*` that they represent, and all `CVMClassBlocks` have an indirect pointer to their `java.lang.Class` instance. This makes it possible to convert between the two by using the following two APIs:

```
extern CVMClassBlock*
CVMgcSafeClassRef2ClassBlock(CVMExecEnv* ee, CVMClassICell *clazz)
extern CVMClassBlock *
CVMgcUnsafeClassRef2ClassBlock(CVMExecEnv *ee, CVMClassICell *clazz)
(gdb) call CVMgcSafeClassRef2ClassBlock(ee, clazz)
$80 = (CVMClassBlock *) 0x26d4d4
```

If `clas` is a `jclass` or `CVMClassICell*`, then you can do the following to print the class name:

```
(gdb) call CVMconsolePrintf("%C\n", CVMgcSafeClassRef2ClassBlock(ee,
clas))
java.lang.Runtime
```

If you are not GC safe, then you can use `CVMgcUnsafeClassRef2ClassBlock()` instead.

See the descriptions of the high-level dumpers in Section A.2.1, "`CVMdumpObject`" on page A-6 and Section A.2.2, "`CVMdumpClassBlock`" on page A-7.

A.5 Other Procedures

A.5.1 Debugging Crashes on Linux

Linux implements threads on top of processes. When there is a crash, the core file produced is almost never for the thread (process) that actually crashed. The CDC-HI virtual machine installs a signal handler on Linux that will catch the signal raised because of a crash, and suspends the process. This allows you to then attach `gdb` to the process, rather than having to deal with a useless core file.

Note that the example below is based on a Linux x86/PC port. The register information will be presented in a different struct for each target.

When the CDC-HI virtual machine crashes on Linux, you will see the following:

```
Process received signal 11, suspending
```

When you see this, `ctrl-z` the process to temporarily stop it. You can then use `ps` to see a list of all the current virtual machine processes:

```
[bin]$ ps

31576 ttyp3 00:00:00 bash
21912 ttyp3 00:00:00 cvm
21913 ttyp3 00:00:00 cvm
21914 ttyp3 00:00:00 cvm
21915 ttyp3 00:00:00 cvm
22447 ttyp3 00:00:00 cvm
22448 ttyp3 00:00:00 cvm
22449 ttyp3 00:00:00 cvm
22450 ttyp3 00:00:00 cvm
22451 ttyp3 00:00:00 cvm
22452 ttyp3 00:00:00 cvm
22453 ttyp3 00:00:00 cvm
22454 ttyp3 00:00:00 cvm
22455 ttyp3 00:00:00 cvm
22456 ttyp3 00:00:00 cvm
22457 ttyp3 00:00:23 cvm
24310 ttyp3 00:00:00 ps
```

The first process in the list is the `main` process and is the one you want to attach to in `gdb`. After executing `ps`, type `bg` to continue execution in the background. Otherwise `gdb` will hang waiting for the process to be started again. Next launch `gdb` and specify the `cvm` binary that was running when the crash occurred.

```
[bin]$ gdb cvm

Current directory is /home/test/cvm/build/linux/bin/

GNU gdb 19991004

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and
you are

welcome to change it and/or distribute copies of it under certain
conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB.  Type "show warranty" for
details.

This GDB was configured as "i386-redhat-linux"...

Starting up GDB configuration file

Done interpreting GDB configuration file
```

Attach to the first CDC-HI virtual machine process in the `ps` list:

```
(gdb) attach 21912

Attaching to program: /home/test/cvm/build/linux/bin/cvm, Pid 21912

Reading symbols from /lib/libpthread.so.0...done.

Reading symbols from /lib/libm.so.6...done.

Reading symbols from /lib/libnsl.so.1...done.

Reading symbols from /lib/libdl.so.2...done.

Reading symbols from /lib/libc.so.6...done.

Reading symbols from /lib/ld-linux.so.2...done.

Reading symbols from /lib/libnss_files.so.2...done.

Reading symbols from /usr/lib/gconv/ISO8859-1.so...done.

0x40083e0b in __sigsuspend (set=0xbffff50c)

    at ../sysdeps/unix/sysv/linux/sigsuspend.c:48

48  ../sysdeps/unix/sysv/linux/sigsuspend.c: No such file or
directory.
```

`info threads` will give you a list of all the threads:

```
(gdb) info threads

  15 Thread 22457  0x40083e0b in __sigsuspend (set=0xbdfff008)

    at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
```

```
14 Thread 22456  0x40083e0b in __sigsuspend (set=0xbe1ff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
13 Thread 22455  0x40083e0b in __sigsuspend (set=0xbe3ff498)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
12 Thread 22454  0x40083e0b in __sigsuspend (set=0xbe5ff498)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
11 Thread 22453  0x40083e0b in __sigsuspend (set=0xbe7ff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
10 Thread 22452  0x40083d61 in __kill () from /lib/libc.so.6
 9 Thread 22451  0x40083e0b in __sigsuspend (set=0xbebff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 8 Thread 22450  0x40083e0b in __sigsuspend (set=0xbedff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 7 Thread 22449  0x40083e0b in __sigsuspend (set=0xbefff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 6 Thread 22447  0x40083e0b in __sigsuspend (set=0xbf1ff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 5 Thread 22448  0x40083e0b in __sigsuspend (set=0xbf3ff62c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 4 Thread 21915  0x40083e0b in __sigsuspend (set=0xbf5ff52c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 3 Thread 21914  0x40083e0b in __sigsuspend (set=0xbf7ff52c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
* 2 Thread 21912 (initial thread)  0x40083e0b in __sigsuspend (set=
0xbffff50c)
   at ../sysdeps/unix/sysv/linux/sigsuspend.c:48
 1 Thread 21913 (manager thread)  0x40110fe0 in __poll (fds=
0x8386d08,
   nfds=1, timeout=2000) at ../sysdeps/unix/sysv/linux/poll.c:45
```

The thread that crashed will be in the `__kill()` function. Switch to it to debug the crash:

```
(gdb) thread 10
[Switching to thread 10 (Thread 22452)]
#0  0x40083d61 in __kill () from /lib/libc.so.6
(gdb) bt
#0  0x40083d61 in __kill () from /lib/libc.so.6
```

```
#1  0x817543d in crash (sig=11) at
../../src/linux/javavm/runtime/sync_md.c:388
#2  0x40022582 in pthread_sighandler (signo=11, ctx={gs = 0, __gsh =
0,
     fs = 0, __fsh = 0, es = 43, __esh = 0, ds = 43, __dsh = 0, edi
= 25,
     esi = 1081082928, ebp = 3198154408, esp = 3198154336, ebx =
1075139692,
     edx = 32, ecx = 1081081904, eax = 1081081912, trapno = 14, err
= 6,
     eip = 1074521900, cs = 35, __csh = 0, eflags = 66118,
     esp_at_signal = 3198154336, ss = 43, __ssh = 0, fpstate =
0xbe9ff5e0,
     oldmask = 2147483648, cr2 = 33}) at signals.c:96
#3  0x40083c88 in __restore ()
    at ../sysdeps/unix/sysv/linux/i386/sigaction.c:127
#4  0x400bfec4 in __libc_calloc (n=1, elem_size=28) at malloc.c:3707
#5  0x8147bb4 in CVMCcallocStub (nelem=1, elsize=28)
    at ../../src/share/javavm/runtime/porting_debug.c:127
#6  0x8114951 in CVMreplenishLockRecordUnsafe (ee=0xbe9ffc40)
    at ../../src/share/javavm/runtime/objsync.c:366
#7  0x8116f59 in CVMdetLock (ee=0xbe9ffc40, indirectObj=0x83d2aec)
    at ../../src/share/javavm/runtime/objsync.c:1116
#8  0x80cc668 in CVMgcUnsafeExecuteJavaMethod (ee=0xbe9ffc40, mb=
0x836a8e4,
    isStatic=0, isVirtual=1)
    at ../../src/share/javavm/runtime/executejava.c:2932
#9  0x80fad05 in CVMjniInvoke (env=0xbe9ffc68, obj=0x83d2954,
    methodID=0x83696e4, pushArguments=0x80fa0d8
<CVMjniPushArgumentsVararg>,
    args=0xbe9ffbdc, info=258, retValue=0x0)
    at ../../src/share/javavm/runtime/jni_impl.c:2412
#10 0x80fc3e1 in CVMjniCallVoidMethod (env=0xbe9ffc68, obj=
0x83d2954,
    methodID=0x83696e4) at
../../src/share/javavm/runtime/jni_impl.c:2587
#11 0x8108830 in start_func (arg=0x409cfc80)
    at ../../src/share/javavm/runtime/jvm.c:1505
#12 0x8173183 in start_func (a=0x409cfc98)
```

```
        at ../../src/portlibs/posix/posix_threads_md.c:30
```

```
#13 0x4001fbb5 in pthread_start_thread (arg=0xbe9ffe40) at
manager.c:241
```

The frame that actually crashed is not included in the backtrace. However, all the
registers, including the pc (eip register) are passed as arguments to the signal
handler, pthread_sighandler(). You can disassemble the value passed in eip to
find out where the crash actually occurred. (Note that not all platforms support this
backtrace feature.)

In the above backtrace, frames #0 through #3 are all part of the signal handling. The
crash actually occurred in a function called from __libc_calloc(). If you
disassemble the value passed in the eip argument to pthread_sighandler(),
you can see that __libc_calloc() called chunk_alloc(), and that is where the
crash occurred. (This crash was the result of a memory corruption that caused a call
to calloc() to crash).

```
(gdb) x /4i 1074521900

0x400be72c <chunk_alloc+84>: mov %ecx,0x8(%edi)

0x400be72f <chunk_alloc+87>: mov 0xfffffff4(%ebp),%edi

0x400be732 <chunk_alloc+90>: orb $0x1,0x4(%edi,%esi,1)

0x400be737 <chunk_alloc+95>: jmp 0x400bef93 <chunk_alloc+2235>
```

NOTE: Some platforms like ARM don't include any frames above the signal handler
in the backtrace.

NOTE: After attaching to the crashed process, you can usually just type continue
and the crash will occur again. But this time gdb will handle the crash and you can
debug at the actual site of the crash rather than in the CDC-HI crash() function.

A.5.2   Debugging Compiled Methods

One way to debug the dynamic compiler with gdb is to use the function
CVMJITcodeCacheFindCompileMethod(<*machine*-pc>, 1). This requires the
CVM_DEBUG=true build. If you know or suspect that a pc is in a compiled method,
you can pass it to the following function:

```
(gdb) call CVMJITcodeCacheFindCompiledMethod(<machine-pc>, 1)
```

If the method is found, it will print out the name of the method and also return the
CVMMethodBlock*. Otherwise NULL is returned.

APPENDIX **B**

C Stack Checking

This appendix describes how system robustness against a tight C stack situation can be achieved by performing static analysis of C stack usage and introducing dynamic stack checks in CDC HI. This appendix covers the following topics:

- Introduction
- Calculating C Stack Redzones
- Worst Cases of Stack Usage Chains

B.1 Introduction

Because CDC HI is targeted for devices that have less memory available and for OSs that do not have MMU support, the C stack usage in CDC HI is taken into account to avoid a stack overrun caused by recursive operations, such as the bytecode interpreter loop, class loading, and class verifier.

To avoid this problem, CDC HI cannot take the same approach as the JDK, in which large memory area is reserved for the C stack. To ensure system robustness against C stack overflow situations caused by tight C stacks, an extensive static analysis of the C codes in CDC HI and the supporting libraries is performed. A dynamic C stack check is introduced at the points where recursive function calls can happen.

As a result, CDC HI guarantees that a C stack overflow failure is detected so that a `StackOverflowError` is thrown when a C stack check notifies that insufficient C stack space is available and a potential memory corruption caused by C stack overflow is eliminated.

C codes are statically analyzed to identify where recursive call cycles occur, eliminate call cycles as much as possible, and sum up stack use in any given call cycles. As a result, the worst case of stack usage required by the recursive function calls is used by a redzone check in the C stack check routine. The worst case of stack usage is called "stack redzone" and the C stack check routine is called

`CVMCstackCheckSize`. The C stack check codes on a cycle determine whether another invocation of the cycle can continue execution with sufficient C stack size left. This redzone (worst case of stack usage) size is used to find out the available stack size for next function invocation in the cycle at execution time. When the C stack is insufficient to continue execution, CDC HI throws a `StackOverflowError`.

B.2   Calculating C Stack Redzones

The CDC HI stack usage is statically analyzed to identify the recursive functions and calculate the worst case C stack redzone values for each recursive function. All CDC HI programs written in C are first compiled into platform-specific assembly codes by the GNU C compiler. All indirect function calls, such as JNI function invocation through the function pointers and other function invocations through function pointers in CDC HI programs, are resolved and mapped into direct function calls to assist the static stack analysis.

The mapping information that maps an indirect function call to a direct function call in CDC HI programs can be found in Chapter 7 and the mapping information that maps an indirect function call to a JNI function call can be found in Chapter 8.

Note – In *stublist*, a function named `CVM_worst_case_psuedo_method` is a link between `CVMjniInvokeNative` and all of the JNI functions in the JNI vector table and the JVMDI vector table. It is also a link between `ansiJavaMain` and all of the JNI functions defined in the JNI vector table for the code outside the VM calling into JNI functions. The worse case stack usage for a JNI call cycle from calling `CVM_worst_case_psuedo_method` to a JNI call is assumed to be 3K bytes, and the same is true for all call cycles that end with OS and C library functions.

There are nine functions that have recursive invocation paths.

1. `CVMgcUnsafeExecuteJavaMethod`

2. `CVMimplementsInterface`

3. `classlookup:CVMclassLookupFromClassLoader`

4. `CVMclassLink`

5. `utils:CVMCstackDummy2CVMformatStringVaList`

6. `utils:CVMCstackDummy2CVMconsolePrintf`

7. `utils:CVMCstackDummy2CVMobjectGetHashSafe`

8. `verifycode:CVMCstackDummy2merge_fullinfo_types`

9. `CVMsignalErrorVaLisT`

Three recursions occurring in the function `CVMformatStringVaList` from file `utils.c` detected by the static stack analysis have at most one-level deep recursion which terminates itself at the second invocation of `CVMformatStringVaList`. To obtain an accurate stack size required to invoke `CVMformatStringVaList` another time, a dummy function is temporarily created for each recursion and is called at the place that needs to check stack overflow before going into the loop to terminate itself. Once the stack requirement is obtained by the static stack analysis, each dummy function invocation is removed and a runtime C stack check is added in the place that invokes the dummy function. There is only one C stack check for detecting the available stack size before calling `utils:CVMCstackDummy2CVMconsolePrintf` and `utils:CVMCstackDummy2CVMobjectGetHashSafe`. The three dummy functions are statically defined in `utils.c` and are invoked in `CVMformatStringVaList` as below:

(1)   `utils:CVMCstackDummy2CVMformatStringVaList`

(2)   `utils:CVMCstackDummy2CVMconsolePrintf`

(3)   `utils:CVMCstackDummy2CVMobjectGetHashSafe`

The self recursive invocation of `merge_fullinfo_types` in file `verifycode.c` has at most one-level deep recursion. A dummy function called `CVMCstackDummy2merge_fullinfo_types` is temporarily created and invoked at the place that needs to check stack overflow before entering the loop and terminating itself in that call path so that the stack redzone can be calculated accurately. Once the stack requirement is obtained by the static stack analysis, this dummy function invocation is removed and a runtime C stack check is added in the place that invokes the dummy function. The dummy function is statically defined in file `verifycode.c` and is invoked in `merge_fullinfo_types` as below:

(1)   `verifycode:CVMCstackDummy2merge_fullinfo_types`

B.2.1   C Stack Redzone Checks

Once the nine recursive functions are identified by the stack analysis, eight stack redzone values are calculated based on the given function call path of each recursion that consumes the most stack usage. The C stack redzone values are used by the C

stack check routines in CDC HI to detect a C stack overflow failure before making a recursive function invocation. The following table shows each C stack redzone macro used by the C stack check in its corresponding recursive function.

TABLE 0-1 C Stack Redzone Macros

CDC HI Function	C Stack Redzone Macro
`CVMgcUnsafeExecuteJavaMethod`	`CVM_REDZONE_ILOOP`
`CVMimplementsInterface`	`CVM_REDZONE_CVMimplementsInterface`
`CVMformatStringVaList`	`CVM_REDZONE_CVMCstackCVMpc2string`
`CVMformatStringVaList`	`CVM_REDZONE_CVMCstackCVMID_objectGet ClassAndHashSafe`
`CVMclassLookupFromClassLoader`	`CVM_REDZONE_CVMclassLookupFromClassL oader`
`CVMclassLink`	`CVM_REDZONE_CVMclassLink`
`merge_fullinfo_to_types`	`CVM_REDZONE_CVMCstackmerge_fullinfo_ to_types`

The following macros are used in C stack check codes for various recursive functions.

TABLE 0-2 C Stack Check Macros

Declaration	Description
`CVM_REDZONE_ILOOP`	Stack space required for interpreter loop in `CVMgcUnsafeExecuteJavaMethod()`.
`CVM_REDZONE_CVMclassLookupFromCla ssLoader`	Stack space required for class loading in a deep class lookup hierarchy in `CVMclassLookupFromClassLoader ()`.
`CVM_REDZONE_CVMclassLink`	Stack space required for class linking in `CVMclassLink()`.
`CVM_REDZONE_CVMclassScan`	Stack space required for traversing the scannable state of one class by the garbage collector in `CVMclassScan()`.
`CVM_REDZONE_CVMimplementsInterfac e`	Stack space required for checking whether a non-array class is an interface type class in `CVMimplementsInterface()`.
`CVM_REDZONE_CVMCstackCVMpc2string`	Stack space required for building a formatted string %P format of PC information for console I/O in `CVMformatStringVaList()`.

TABLE 0-2   C Stack Check Macros *(Continued)*

Declaration	Description
`CVM_REDZONE_CVMCstackCVMID_` `objectGetClassAndHashSafe`	Stack space required for building a formatted string %I format of class name information for console I/O in `CVMformatStringVaList()`.
`CVM_REDZONE_CVMCstackmerge_` `fullinfo_to_types`	Stack space required for class verifier to do type merging between two object types or two arrays of object types in `merge_fullinfo_types()`.
`CVM_REDZONE_CVMsignalErrorVaList`	Stack space required for signaling an exception in `CVMsignalErrorVaList()`.

B.2.1.1   Recursive Functions

Recursive functions that have a C stack check are listed below:

- `CVMgcUnsafeExecuteJavaMethod`
- `CVMclassLookupFromClassLoader`
- `CVMclassLink`
- `CVMclassScan`
- `CVMimplementsInterface`
- `CVMformatStringVaList`
- `CVMsignalErrorVaList`
- `merge_fullinfo_types`

B.2.2   C Stack Redzone Values

The following eight stack redzones are used by the stack check routines in CDC HI are shown in the table below.

TABLE 0-3   C Stack Redzone Macro Values

C Stack Redzone Macro	Stack Redzone Value
`CVM_REDZONE_ILOOP`	11592 (11.32K bytes)
`CVM_REDZONE_CVMimplementsInterface`	4752 (5.14K bytes)
`CVM_REDZONE_CVMCstackCVMpc2string`	4752 (5.14K bytes)

TABLE 0-3   C Stack Redzone Macro Values

C Stack Redzone Macro	Stack Redzone Value
`CVM_REDZONE_CVMCstackCVMID_objectGetClassAndHashSafe`	5392 (5.27K bytes)
`CVM_REDZONE_CVMclassLookupFromClassLoader`	13500 (13.18K bytes)
`CVM_REDZONE_CVMclassLink`	7456 (7.28K bytes)
`CVM_REDZONE_CVMCstackmerge_fullinfo_to_types`	5800 (5.66K bytes)
`CVM_REDZONE_CVMsignalErrorVaList`	8400 (8.20K bytes)

B.3   Worst Cases of Stack Usage Chains

The worst cases of stack usage chains that are used to calculate the stack redzones can vary greatly on different platforms. SPARC, a RISC platform, uses a minimum 112 bytes of frame allocation in a frame even if no local variables are declared (in some special case, code optimizer can make this 0). Whereas, many CISC platforms would only use less stack space in this case. Therefore, stack usage calculated based on SPARC code tends to be larger than CISC platforms.

The worst stack usage chain that contains the stack size of each function and the accumulated stack size of the call path is not necessarily one of the recursive paths. It can be a stack trace of calling a recursive function and terminating at an OS or C library function, a stack trace of calling a recursive function and terminating at a CDC HI function that ends itself, or a stack trace of calling a self-recursive function. The static stack analysis checks all the call paths and comes out with the worst stack usage of each recursive function call cycle. All the worst C stack usage call chains provided in this documentation terminate at a library function `strncpy`.

The static stack analysis looks into the worst case of the stack usage chain of each recursive function cycle and identifies several functions that use more stack frame space than other functions occurring in the stack usage chain. The functions that consume a large chunk of stack space are optimized by replacing the static allocation of locals with a dynamically allocated and shared buffer to reduce the stack usage of the frame allocation. Other optimizations summarized by the static stack analysis include removing a recursion or a potentially infinite recursion out of the recursive functions to reduce the stack space usage.

Listed in the following sections are the worst stack usage chains. They trace function call paths that most consume the stack in each recursive function.

The worst stack usage chains are:

- Interpreter Loop
- Implement Interface
- Class Lookup in Class Loader
- Class Link
- Format String from PC
- Format String from Object
- Class Verifier
- Signal Exception Message

In the following sections, the worst stack usage lists are structured as follows:

- The first number is the accumulated stack usage in bytes to execute functions from bottom up to the point that excludes the stack size of the current function specified in the same line.
- The second number is the stack frame size of the function following the second colon.

B.3.1   Worst C Stack Usage Chain in Interpreter Loop

Some worst cases of stack consumption occurring in the interpreter loop that have a call path from `CVMgcUnsafeExecuteJavaMethod` to a C library function, such as `strncpy`, are given below.

```
> stacktrace CVMgcUnsafeExecuteJavaMethod strncpy
 8848   : 328   : CVMgcUnsafeExecuteJavaMethod
 8520   : 112   : CVMjniInvokeNative
 8408   : 3072  : CVM_worst_case_psuedo_method
 5336   : 128   : CVMjniToReflectedMethod
 5208   : 112   : CVMreflectMethodBlockToNewJavaMethod
 5096   : 224   : CVMreflectNewJavaLangReflectMethod
 4872   : 376   : CVMnewStringUTF
 4496   : 144   : CVMnewString
 4352   : 136   : CVMgcAllocNewInstance
 4216   : 128   : CVMjniCallStaticVoidMethod
 4088   : 184   : jni_impl:CVMjniInvoke
 3904   : 136   : CVMjniExceptionDescribe
 3768   : 112   : CVMjniGetMethodID
 3656   : 136   : jni_impl:CVMjniGetXMethodID
 3520   : 112   : CVMclassInit
 3408   : 128   : classinitialize:CVMprivateClassInit
```

```
3280    : 128    : CVMjniCallVoidMethod
3152    : 184    : jni_impl:CVMjniInvoke
2968    : 136    : CVMjniExceptionDescribe
2832    : 112    : CVMjniGetMethodID
2720    : 136    : jni_impl:CVMjniGetXMethodID
2584    : 128    : CVMtypeidNewMethodIDFromNameAndSig
2456    : 240    : typeid:referenceMethodSignature
2216    : 144    : typeid:referenceFieldSignature
2072    : 120    : typeid:referenceClassName
1952    : 144    : typeid:lookupClass
1808    : 120    : typeid:findFreeScalarEntry
1688    : 136    : typeid:findFreeTableEntry
1552    : 120    : typeid:unlockThrowOutOfMemoryError
1432    : 112    : CVMsysMutexUnlock
1320    : 376    : CVMconsolePrintf
944     : 304    : CVMformatStringVaList
640     : 168    : CVMtypeidMethodTypeToCString
472     : 112    : CVMtypeidFieldTypeToCString
360     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
232     : 120    : typeid:conditionalPutstring
112     : 112    : CVMCstrncpyStub
0       : ?(0)   : strncpy

> stacktrace CVMgcUnsafeExecuteJavaMethod .umul
8744    : 328    : CVMgcUnsafeExecuteJavaMethod
8416    : 112    : CVMjniInvokeNative
8304    : 3072   : CVM_worst_case_psuedo_method
5232    : 128    : CVMjniToReflectedMethod
5104    : 112    : CVMreflectMethodBlockToNewJavaMethod
4992    : 224    : CVMreflectNewJavaLangReflectMethod
4768    : 376    : CVMnewStringUTF
4392    : 144    : CVMnewString
4248    : 136    : CVMgcAllocNewInstance
4112    : 128    : CVMjniCallStaticVoidMethod
3984    : 184    : jni_impl:CVMjniInvoke
3800    : 136    : CVMjniExceptionDescribe
```

```
3664   : 112   : CVMjniGetMethodID
3552   : 136   : jni_impl:CVMjniGetXMethodID
3416   : 112   : CVMclassInit
3304   : 128   : classinitialize:CVMprivateClassInit
3176   : 128   : CVMjniCallVoidMethod
3048   : 184   : jni_impl:CVMjniInvoke
2864   : 136   : CVMjniExceptionDescribe
2728   : 112   : CVMjniGetMethodID
2616   : 136   : jni_impl:CVMjniGetXMethodID
2480   : 128   : CVMtypeidNewMethodIDFromNameAndSig
2352   : 240   : typeid:referenceMethodSignature
2112   : 144   : typeid:referenceFieldSignature
1968   : 120   : typeid:referenceClassName
1848   : 144   : typeid:lookupClass
1704   : 120   : typeid:findFreeScalarEntry
1584   : 136   : typeid:findFreeTableEntry
1448   : 120   : typeid:unlockThrowOutOfMemoryError
1328   : 112   : CVMsysMutexUnlock
1216   : 376   : CVMconsolePrintf
840    : 304   : CVMformatStringVaList
536    : 168   : CVMtypeidMethodTypeToCString
368    : 128   : CVMtypeidGetSignatureIterator
240    : 120   : CVMtypeidGetTerseSignatureIterator
120    : 120   : typeid:indexSegmentedTable
0      : ?(0)  : .umul

> stacktrace CVMgcUnsafeExecuteJavaMethod strlen
 8728   : 328   : CVMgcUnsafeExecuteJavaMethod
 8400   : 112   : CVMjniInvokeNative
 8288   : 3072  : CVM_worst_case_psuedo_method
 5216   : 128   : CVMjniToReflectedMethod
 5088   : 112   : CVMreflectMethodBlockToNewJavaMethod
 4976   : 224   : CVMreflectNewJavaLangReflectMethod
 4752   : 376   : CVMnewStringUTF
 4376   : 144   : CVMnewString
 4232   : 136   : CVMgcAllocNewInstance
```

```
4096   : 128    : CVMjniCallStaticVoidMethod
3968   : 184    : jni_impl:CVMjniInvoke
3784   : 136    : CVMjniExceptionDescribe
3648   : 112    : CVMjniGetMethodID
3536   : 136    : jni_impl:CVMjniGetXMethodID
3400   : 112    : CVMclassInit
3288   : 128    : classinitialize:CVMprivateClassInit
3160   : 128    : CVMjniCallVoidMethod
3032   : 184    : jni_impl:CVMjniInvoke
2848   : 136    : CVMjniExceptionDescribe
2712   : 112    : CVMjniGetMethodID
2600   : 136    : jni_impl:CVMjniGetXMethodID
2464   : 128    : CVMtypeidNewMethodIDFromNameAndSig
2336   : 240    : typeid:referenceMethodSignature
2096   : 144    : typeid:referenceFieldSignature
1952   : 120    : typeid:referenceClassName
1832   : 144    : typeid:lookupClass
1688   : 120    : typeid:findFreeScalarEntry
1568   : 136    : typeid:findFreeTableEntry
1432   : 120    : typeid:unlockThrowOutOfMemoryError
1312   : 112    : CVMsysMutexUnlock
1200   : 376    : CVMconsolePrintf
824    : 304    : CVMformatStringVaList
520    : 168    : CVMtypeidMethodTypeToCString
352    : 112    : CVMtypeidFieldTypeToCString
240    : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
112    : 112    : CVMCstrlenStub
0      : ?(0)   : strlen


> stacktrace CVMgcUnsafeExecuteJavaMethod free
8320   : 328    : CVMgcUnsafeExecuteJavaMethod
7992   : 112    : CVMjniInvokeNative
7880   : 3072   : CVM_worst_case_psuedo_method
4808   : 128    : CVMjniToReflectedMethod
4680   : 112    : CVMreflectMethodBlockToNewJavaMethod
```

```
4568    : 224    : CVMreflectNewJavaLangReflectMethod
4344    : 376    : CVMnewStringUTF
3968    : 144    : CVMnewString
3824    : 136    : CVMgcAllocNewInstance
3688    : 128    : CVMjniCallStaticVoidMethod
3560    : 184    : jni_impl:CVMjniInvoke
3376    : 136    : CVMjniExceptionDescribe
3240    : 112    : CVMjniGetMethodID
3128    : 136    : jni_impl:CVMjniGetXMethodID
2992    : 112    : CVMclassInit
2880    : 128    : classinitialize:CVMprivateClassInit
2752    : 128    : CVMjniCallVoidMethod
2624    : 184    : jni_impl:CVMjniInvoke
2440    : 136    : CVMjniExceptionDescribe
2304    : 112    : CVMjniGetMethodID
2192    : 136    : jni_impl:CVMjniGetXMethodID
2056    : 128    : CVMtypeidNewMethodIDFromNameAndSig
1928    : 240    : typeid:referenceMethodSignature
1688    : 144    : typeid:referenceFieldSignature
1544    : 120    : typeid:referenceClassName
1424    : 144    : typeid:lookupClass
1280    : 120    : typeid:findFreeScalarEntry
1160    : 136    : typeid:findFreeTableEntry
1024    : 120    : typeid:unlockThrowOutOfMemoryError
904     : 112    : CVMsysMutexUnlock
792     : 376    : CVMconsolePrintf
416     : 304    : CVMformatStringVaList
112     : 112    : CVMCfreeStub
0       : ?(0)   : free

> stacktrace CVMgcUnsafeExecuteJavaMethod sprintf
8208    : 328    : CVMgcUnsafeExecuteJavaMethod
7880    : 112    : CVMjniInvokeNative
7768    : 3072   : CVM_worst_case_psuedo_method
4696    : 128    : CVMjniToReflectedMethod
4568    : 112    : CVMreflectMethodBlockToNewJavaMethod
```

```
4456    : 224     : CVMreflectNewJavaLangReflectMethod
4232    : 376     : CVMnewStringUTF
3856    : 144     : CVMnewString
3712    : 136     : CVMgcAllocNewInstance
3576    : 128     : CVMjniCallStaticVoidMethod
3448    : 184     : jni_impl:CVMjniInvoke
3264    : 136     : CVMjniExceptionDescribe
3128    : 112     : CVMjniGetMethodID
3016    : 136     : jni_impl:CVMjniGetXMethodID
2880    : 112     : CVMclassInit
2768    : 128     : classinitialize:CVMprivateClassInit
2640    : 128     : CVMjniCallVoidMethod
2512    : 184     : jni_impl:CVMjniInvoke
2328    : 136     : CVMjniExceptionDescribe
2192    : 112     : CVMjniGetMethodID
2080    : 136     : jni_impl:CVMjniGetXMethodID
1944    : 128     : CVMtypeidNewMethodIDFromNameAndSig
1816    : 240     : typeid:referenceMethodSignature
1576    : 144     : typeid:referenceFieldSignature
1432    : 120     : typeid:referenceClassName
1312    : 144     : typeid:lookupClass
1168    : 120     : typeid:findFreeScalarEntry
1048    : 136     : typeid:findFreeTableEntry
912     : 120     : typeid:unlockThrowOutOfMemoryError
792     : 112     : CVMsysMutexUnlock
680     : 376     : CVMconsolePrintf
304     : 304     : CVMformatStringVaList
0       : ?(0)    : sprintf

> stacktrace CVMgcUnsafeExecuteJavaMethod fflush
7904    : 328     : CVMgcUnsafeExecuteJavaMethod
7576    : 112     : CVMjniInvokeNative
7464    : 3072    : CVM_worst_case_psuedo_method
4392    : 128     : CVMjniToReflectedMethod

4264    : 112     : CVMreflectMethodBlockToNewJavaMethod
```

```
4152   : 224    : CVMreflectNewJavaLangReflectMethod

3928   : 376    : CVMnewStringUTF
3552   : 144    : CVMnewString
3408   : 136    : CVMgcAllocNewInstance
3272   : 128    : CVMjniCallStaticVoidMethod
3144   : 184    : jni_impl:CVMjniInvoke
2960   : 136    : CVMjniExceptionDescribe
2824   : 112    : CVMjniGetMethodID
2712   : 136    : jni_impl:CVMjniGetXMethodID
2576   : 112    : CVMclassInit
2464   : 128    : classinitialize:CVMprivateClassInit
2336   : 128    : CVMjniCallVoidMethod
2208   : 184    : jni_impl:CVMjniInvoke
2024   : 136    : CVMjniExceptionDescribe
1888   : 112    : CVMjniGetMethodID
1776   : 136    : jni_impl:CVMjniGetXMethodID

1640   : 128    : CVMtypeidNewMethodIDFromNameAndSig
1512   : 240    : typeid:referenceMethodSignature
1272   : 144    : typeid:referenceFieldSignature
1128   : 120    : typeid:referenceClassName
1008   : 144    : typeid:lookupClass
864    : 120    : typeid:findFreeScalarEntry
744    : 136    : typeid:findFreeTableEntry
608    : 120    : typeid:unlockThrowOutOfMemoryError
488    : 112    : CVMsysMutexUnlock
376    : 376    : CVMconsolePrintf
0      : ?(0)   : fflush

> stacktrace CVMgcUnsafeExecuteJavaMethod .rem
 7424   : 328    : CVMgcUnsafeExecuteJavaMethod
 7096   : 112    : CVMjniInvokeNative
 6984   : 3072   : CVM_worst_case_psuedo_method
 3912   : 128    : CVMjniToReflectedMethod
 3784   : 112    : CVMreflectMethodBlockToNewJavaMethod
```

```
3672    : 224    : CVMreflectNewJavaLangReflectMethod
3448    : 376    : CVMnewStringUTF
3072    : 144    : CVMnewString
2928    : 136    : CVMgcAllocNewInstance
2792    : 128    : CVMjniCallStaticVoidMethod
2664    : 184    : jni_impl:CVMjniInvoke
2480    : 136    : CVMjniExceptionDescribe
2344    : 112    : CVMjniGetMethodID
2232    : 136    : jni_impl:CVMjniGetXMethodID
2096    : 112    : CVMclassInit
1984    : 128    : classinitialize:CVMprivateClassInit
1856    : 128    : CVMjniCallVoidMethod
1728    : 184    : jni_impl:CVMjniInvoke
1544    : 136    : CVMjniExceptionDescribe
1408    : 128    : CVMjniGetObjectClass
1280    : 120    : CVMjniNewLocalRef
1160    : 144    : CVMjniCreateLocalRef
1016    : 112    : CVMjniFatalError
904     : 120    : CVMdumpStack
784     : 128    : stacks:CVMdumpFrames
656     : 392    : CVMdumpFrame
264     : 128    : CVMpc2string
136     : 136    : interpreter:CVMadddec
0       : ?(0)   : .rem
```

B.3.2 Worst C Stack Usage Chain in `CVMimplementsInterface` Routine

Some worst cases of stack consumption occurring in the routine that compares each subclass class block in the class hierarchy to the interface class block have a call path from `CVMimplementsInterface` to a C library function, such as `strncpy`, are given below.

```
> stacktrace CVMimplementsInterface strncpy


1800    : 120    : CVMimplementsInterface
1680    : 112    : CVMCstackCheckSize
```

```
1568    : 136    : CVMcsRendezvous
1432    : 112    : CVMsysMutexLock
1320    : 376    : CVMconsolePrintf
944     : 304    : CVMformatStringVaList
640     : 168    : CVMtypeidMethodTypeToCString
472     : 112    : CVMtypeidFieldTypeToCString
360     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
232     : 120    : typeid:conditionalPutstring
112     : 112    : CVMCstrncpyStub
0       : ?(0)   : strncpy

> stacktrace CVMimplementsInterface .umul
 1696   : 120    : CVMimplementsInterface
 1576   : 112    : CVMCstackCheckSize
 1464   : 136    : CVMcsRendezvous
 1328   : 112    : CVMsysMutexLock
 1216   : 376    : CVMconsolePrintf
 840    : 304    : CVMformatStringVaList
 536    : 168    : CVMtypeidMethodTypeToCString
 368    : 128    : CVMtypeidGetSignatureIterator
 240    : 120    : CVMtypeidGetTerseSignatureIterator
 120    : 120    : typeid:indexSegmentedTable
 0      : ?(0)   : .umul

> stacktrace CVMimplementsInterface strlen
 1680   : 120    : CVMimplementsInterface
 1560   : 112    : CVMCstackCheckSize
 1448   : 136    : CVMcsRendezvous
 1312   : 112    : CVMsysMutexLock
 1200   : 376    : CVMconsolePrintf
 824    : 304    : CVMformatStringVaList
 520    : 168    : CVMtypeidMethodTypeToCString
 352    : 112    : CVMtypeidFieldTypeToCString
 240    : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
 112    : 112    : CVMCstrlenStub
 0      : ?(0)   : strlen
```

```
> stacktrace CVMimplementsInterface free
 1272   : 120   : CVMimplementsInterface
 1152   : 112   : CVMCstackCheckSize
 1040   : 136   : CVMcsRendezvous
 904    : 112   : CVMsysMutexLock
 792    : 376   : CVMconsolePrintf
 416    : 304   : CVMformatStringVaList
 112    : 112   : CVMCfreeStub
 0      : ?(0)  : free


> stacktrace CVMimplementsInterface sprintf
 1160   : 120   : CVMimplementsInterface
 1040   : 112   : CVMCstackCheckSize
 928    : 136   : CVMcsRendezvous
 792    : 112   : CVMsysMutexLock
 680    : 376   : CVMconsolePrintf
 304    : 304   : CVMformatStringVaList
 0      : ?(0)  : sprintf


> stacktrace CVMimplementsInterface pthread_cond_wait
 936    : 120   : CVMimplementsInterface
 816    : 112   : CVMCstackCheckSize
 704    : 136   : CVMcsRendezvous
 568    : 128   : CVMsysMutexWait
 440    : 128   : CVMreentrantMutexWait
 312    : 144   : CVMcondvarWait
 168    : 168   : POSIXcondvarWait
 0      : ?(0)  : pthread_cond_wait
```

B.3.3 Worst C Stack Usage Chain in Class Loader at Class Lookup

Some worst cases of stack consumption occurs in class loader that have a call path from CVMclassLookupFromClassLoader to a C library function such as strncpy are given below.

```
> stacktrace classlookup:CVMclassLookupFromClassLoader strncpy
 5968   : 152   : classlookup:CVMclassLookupFromClassLoader
 5816   : 136   : CVMclassLoadClass
 5680   : 136   : classload:CVMclassLoadSystemClass
 5544   : 144   : classload:CVMclassLoadFromFile
 5400   : 672   : CVMclassCreateInternalClass
 4728   : 168   : classcreate:CVMreadConstantPool
 4560   : 376   : CVMinternUTF
 4184   : 136   : CVMgcAllocNewInstance
 4048   : 136   : CVMgcimplAllocObject
 3912   : 120   : CVMgcStopTheWorldAndGC
 3792   : 128   : gc_common:CVMgcStopTheWorldAndGCSafe
 3664   : 144   : CVMgcimplDoGC
 3520   : 112   : gen_semispace:CVMgenSemispaceCollect
 3408   : 144   : gen_semispace:CVMgenSemispaceCollectFull
 3264   : 112   : CVMgcProcessSpecialWithLivenessInfo
 3152   : 136   : CVMweakrefProcessNonStrong
 3016   : 128   : weakrefs:CVMweakrefHandlePendingQueue
 2888   : 120   : weakrefs:CVMweakrefIterateQueue
 2768   : 136   : weakrefs:CVMweakrefDiscoveredQueueCallback
 2632   : 128   : gen_semispace:CVMgenSemispaceScanTransitively
 2504   : 112   : gen_semispace:CVMgenSemispaceFollowRootsFull
 2392   : 136   :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
 2256   : 120   : gen_semispace:CVMgenSemispaceScanPromotedPointers
 2136   : 168   : gen_semispace:scanObjectsInRangeFull
 1968   : 152   : gen_semispace:CVMgenSemispaceBlackenObjectFull
 1816   : 112   : gen_semispace:CVMgenSemispaceFilteredGrayObject
 1704   : 120   : gen_semispace:CVMgenSemispaceGrayObject
 1584   : 136   : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
 1448   : 128   : gen_semispace:CVMgenSemispacePromoteInto
 1320   : 376   : CVMconsolePrintf
  944   : 304   : CVMformatStringVaList
  640   : 168   : CVMtypeidMethodTypeToCString
  472   : 112   : CVMtypeidFieldTypeToCString
  360   : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
```

```
 232     : 120    : typeid:conditionalPutstring
 112     : 112    : CVMCstrncpyStub
   0     : ?(0)   : strncpy


> stacktrace classlookup:CVMclassLookupFromClassLoader .umul
5864   : 152    : classlookup:CVMclassLookupFromClassLoader
5712   : 136    : CVMclassLoadClass
5576   : 136    : classload:CVMclassLoadSystemClass
5440   : 144    : classload:CVMclassLoadFromFile
5296   : 672    : CVMclassCreateInternalClass
4624   : 168    : classcreate:CVMreadConstantPool
4456   : 376    : CVMinternUTF
4080   : 136    : CVMgcAllocNewInstance
3944   : 136    : CVMgcimplAllocObject
3808   : 120    : CVMgcStopTheWorldAndGC
3688   : 128    : gc_common:CVMgcStopTheWorldAndGCSafe
3560   : 144    : CVMgcimplDoGC
3416   : 112    : gen_semispace:CVMgenSemispaceCollect
3304   : 144    : gen_semispace:CVMgenSemispaceCollectFull
3160   : 112    : CVMgcProcessSpecialWithLivenessInfo
3048   : 136    : CVMweakrefProcessNonStrong
2912   : 128    : weakrefs:CVMweakrefHandlePendingQueue
2784   : 120    : weakrefs:CVMweakrefIterateQueue
2664   : 136    : weakrefs:CVMweakrefDiscoveredQueueCallback
2528   : 128    : gen_semispace:CVMgenSemispaceScanTransitively
2400   : 112    : gen_semispace:CVMgenSemispaceFollowRootsFull
2288   : 136    :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
2152   : 120    : gen_semispace:CVMgenSemispaceScanPromotedPointers
2032   : 168    : gen_semispace:scanObjectsInRangeFull
1864   : 152    : gen_semispace:CVMgenSemispaceBlackenObjectFull
1712   : 112    : gen_semispace:CVMgenSemispaceFilteredGrayObject
1600   : 120    : gen_semispace:CVMgenSemispaceGrayObject
1480   : 136    : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
1344   : 128    : gen_semispace:CVMgenSemispacePromoteInto
1216   : 376    : CVMconsolePrintf
```

```
840     : 304    : CVMformatStringVaList
536     : 168    : CVMtypeidMethodTypeToCString
368     : 128    : CVMtypeidGetSignatureIterator
240     : 120    : CVMtypeidGetTerseSignatureIterator
120     : 120    : typeid:indexSegmentedTable
0       : ?(0)   : .umul


> stacktrace classlookup:CVMclassLookupFromClassLoader strlen
5848    : 152    : classlookup:CVMclassLookupFromClassLoader
5696    : 136    : CVMclassLoadClass
5560    : 136    : classload:CVMclassLoadSystemClass
5424    : 144    : classload:CVMclassLoadFromFile
5280    : 672    : CVMclassCreateInternalClass
4608    : 168    : classcreate:CVMreadConstantPool
4440    : 376    : CVMinternUTF
4064    : 136    : CVMgcAllocNewInstance
3928    : 136    : CVMgcimplAllocObject
3792    : 120    : CVMgcStopTheWorldAndGC
3672    : 128    : gc_common:CVMgcStopTheWorldAndGCSafe
3544    : 144    : CVMgcimplDoGC
3400    : 112    : gen_semispace:CVMgenSemispaceCollect
3288    : 144    : gen_semispace:CVMgenSemispaceCollectFull
3144    : 112    : CVMgcProcessSpecialWithLivenessInfo
3032    : 136    : CVMweakrefProcessNonStrong
2896    : 128    : weakrefs:CVMweakrefHandlePendingQueue
2768    : 120    : weakrefs:CVMweakrefIterateQueue
2648    : 136    : weakrefs:CVMweakrefDiscoveredQueueCallback
2512    : 128    : gen_semispace:CVMgenSemispaceScanTransitively
2384    : 112    : gen_semispace:CVMgenSemispaceFollowRootsFull
2272    : 136    :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
2136    : 120    : gen_semispace:CVMgenSemispaceScanPromotedPointers
2016    : 168    : gen_semispace:scanObjectsInRangeFull
1848    : 152    : gen_semispace:CVMgenSemispaceBlackenObjectFull
1696    : 112    : gen_semispace:CVMgenSemispaceFilteredGrayObject
1584    : 120    : gen_semispace:CVMgenSemispaceGrayObject
```

```
1464   : 136   : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
1328   : 128   : gen_semispace:CVMgenSemispacePromoteInto
1200   : 376   : CVMconsolePrintf
824    : 304   : CVMformatStringVaList
520    : 168   : CVMtypeidMethodTypeToCString
352    : 112   : CVMtypeidFieldTypeToCString
240    : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
112    : 112   : CVMCstrlenStub
0      : ?(0)  : strlen


> stacktrace classlookup:CVMclassLookupFromClassLoader adler32
5568   : 152   : classlookup:CVMclassLookupFromClassLoader
5416   : 136   : CVMclassLoadClass
5280   : 136   : classload:CVMclassLoadSystemClass
5144   : 136   : classload:CVMclassLoadFromZipFile
5008   : 128   : CVMziputilReadEntry
4880   : 4280  : CVMzutilInflateFully
600    : 120   : CVMzlibInflate
480    : 192   : CVMzlibInflate_blocks
288    : 160   : CVMzlibInflate_codes
128    : 128   : CVMzlibInflate_flush
0      : ?(0)  : adler32


> stacktrace classlookup:CVMclassLookupFromClassLoader
pthread_getspecific
5560   : 152   : classlookup:CVMclassLookupFromClassLoader
5408   : 136   : CVMclassLoadClass
5272   : 136   : classload:CVMclassLoadSystemClass
5136   : 144   : classload:CVMclassLoadFromFile
4992   : 672   : CVMclassCreateInternalClass
4320   : 168   : classcreate:CVMreadConstantPool
4152   : 376   : CVMinternUTF
3776   : 136   : CVMgcAllocNewInstance
3640   : 136   : CVMgcimplAllocObject
3504   : 120   : CVMgcStopTheWorldAndGC
3384   : 128   : gc_common:CVMgcStopTheWorldAndGCSafe
```

```
3256    : 144    : CVMgcimplDoGC
3112    : 112    : gen_semispace:CVMgenSemispaceCollect
3000    : 144    : gen_semispace:CVMgenSemispaceCollectFull
2856    : 112    : CVMgcProcessSpecialWithLivenessInfo
2744    : 136    : CVMweakrefProcessNonStrong
2608    : 128    : weakrefs:CVMweakrefHandlePendingQueue
2480    : 120    : weakrefs:CVMweakrefIterateQueue
2360    : 136    : weakrefs:CVMweakrefDiscoveredQueueCallback
2224    : 128    : gen_semispace:CVMgenSemispaceScanTransitively
2096    : 112    : gen_semispace:CVMgenSemispaceFollowRootsFull
1984    : 136    :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
1848    : 120    : gen_semispace:CVMgenSemispaceScanPromotedPointers
1728    : 168    : gen_semispace:scanObjectsInRangeFull
1560    : 152    : gen_semispace:CVMgenSemispaceBlackenObjectFull
1408    : 112    : gen_semispace:CVMgenSemispaceFilteredGrayObject
1296    : 120    : gen_semispace:CVMgenSemispaceGrayObject
1176    : 136    : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
1040    : 128    : gen_semispace:CVMgenSemispacePromoteInto
912     : 376    : CVMconsolePrintf
536     : 304    : CVMformatStringVaList
232     : 120    : CVMgetEE
112     : 112    : POSIXthreadGetSelf
0       : ?(0)   : pthread_getspecific

> stacktrace classlookup:CVMclassLookupFromClassLoader zcfree
5528    : 152    : classlookup:CVMclassLookupFromClassLoader
5376    : 136    : CVMclassLoadClass
5240    : 136    : classload:CVMclassLoadSystemClass
5104    : 136    : classload:CVMclassLoadFromZipFile
4968    : 128    : CVMziputilReadEntry
4840    : 4280   : CVMzutilInflateFully
560     : 112    : CVMzlibInflateInit2_
448     : 112    : CVMzlibInflateEnd
336     : 112    : CVMzlibInflate_blocks_free
224     : 112    : CVMzlibInflate_blocks_reset
```

```
    112    : 112    : CVMzlibInflate_codes_free
    0      : ?(0)    : zcfree


> stacktrace classlookup:CVMclassLookupFromClassLoader free
  5440   : 152    : classlookup:CVMclassLookupFromClassLoader
  5288   : 136    : CVMclassLoadClass
  5152   : 136    : classload:CVMclassLoadSystemClass
  5016   : 144    : classload:CVMclassLoadFromFile
  4872   : 672    : CVMclassCreateInternalClass
  4200   : 168    : classcreate:CVMreadConstantPool
  4032   : 376    : CVMinternUTF
  3656   : 136    : CVMgcAllocNewInstance
  3520   : 136    : CVMgcimplAllocObject
  3384   : 120    : CVMgcStopTheWorldAndGC
  3264   : 128    : gc_common:CVMgcStopTheWorldAndGCSafe
  3136   : 144    : CVMgcimplDoGC
  2992   : 112    : gen_semispace:CVMgenSemispaceCollect
  2880   : 144    : gen_semispace:CVMgenSemispaceCollectFull
  2736   : 112    : CVMgcProcessSpecialWithLivenessInfo
  2624   : 136    : CVMweakrefProcessNonStrong
  2488   : 128    : weakrefs:CVMweakrefHandlePendingQueue
  2360   : 120    : weakrefs:CVMweakrefIterateQueue
  2240   : 136    : weakrefs:CVMweakrefDiscoveredQueueCallback
  2104   : 128    : gen_semispace:CVMgenSemispaceScanTransitively
  1976   : 112    : gen_semispace:CVMgenSemispaceFollowRootsFull
  1864   : 136    :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
  1728   : 120    : gen_semispace:CVMgenSemispaceScanPromotedPointers
  1608   : 168    : gen_semispace:scanObjectsInRangeFull
  1440   : 152    : gen_semispace:CVMgenSemispaceBlackenObjectFull
  1288   : 112    : gen_semispace:CVMgenSemispaceFilteredGrayObject
  1176   : 120    : gen_semispace:CVMgenSemispaceGrayObject
  1056   : 136    : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
  920    : 128    : gen_semispace:CVMgenSemispacePromoteInto
  792    : 376    : CVMconsolePrintf
  416    : 304    : CVMformatStringVaList
```

```
112     : 112    : CVMCfreeStub
0       : ?(0)    : free


> stacktrace classlookup:CVMclassLookupFromClassLoader strchr
 5440    : 152    : classlookup:CVMclassLookupFromClassLoader
 5288    : 136    : CVMclassLoadClass
 5152    : 136    : classload:CVMclassLoadSystemClass
 5016    : 144    : classload:CVMclassLoadFromFile
 4872    : 672    : CVMclassCreateInternalClass
 4200    : 168    : classcreate:CVMreadConstantPool
 4032    : 376    : CVMinternUTF
 3656    : 136    : CVMgcAllocNewInstance
 3520    : 136    : CVMgcimplAllocObject
 3384    : 120    : CVMgcStopTheWorldAndGC
 3264    : 128    : gc_common:CVMgcStopTheWorldAndGCSafe
 3136    : 144    : CVMgcimplDoGC
 2992    : 112    : gen_semispace:CVMgenSemispaceCollect
 2880    : 144    : gen_semispace:CVMgenSemispaceCollectFull
 2736    : 112    : CVMgcProcessSpecialWithLivenessInfo
 2624    : 136    : CVMweakrefProcessNonStrong
 2488    : 128    : weakrefs:CVMweakrefHandlePendingQueue
 2360    : 120    : weakrefs:CVMweakrefIterateQueue
 2240    : 136    : weakrefs:CVMweakrefDiscoveredQueueCallback
 2104    : 128    : gen_semispace:CVMgenSemispaceScanTransitively
 1976    : 112    : gen_semispace:CVMgenSemispaceFollowRootsFull
 1864    : 136    :
gen_semispace:CVMgenSemispaceFollowRootsWithBlackener
 1728    : 120    : gen_semispace:CVMgenSemispaceScanPromotedPointers
 1608    : 168    : gen_semispace:scanObjectsInRangeFull
 1440    : 152    : gen_semispace:CVMgenSemispaceBlackenObjectFull
 1288    : 112    : gen_semispace:CVMgenSemispaceFilteredGrayObject
 1176    : 120    : gen_semispace:CVMgenSemispaceGrayObject
 1056    : 136    : gen_semispace:CVMgenSemispaceForwardOrPromoteObject
 920     : 128    : gen_semispace:CVMgenSemispacePromoteInto
 792     : 376    : CVMconsolePrintf
 416     : 304    : CVMformatStringVaList
```

```
112     : 112   : CVMCstrchrStub
0       : ?(0)   : strchr
```

B.3.4   Worst C Stack Usage Chain in Class Link

Some worst cases of stack consumption occurring in class linking that have a call
path from `CVMclassLink` to a C library function such as `strncpy` are given below.

```
> stacktrace CVMclassLink strncpy
 4512   : 128   : CVMclassLink
 4384   : 376   : CVMclassVerify
 4008   : 144   : VerifyClass
 3864   : 160   : verifycode:verify_method
 3704   : 184   : verifycode:run_dataflow
 3520   : 200   : verifycode:merge_into_successors

 3320   : 264   : verifycode:merge_into_one_successor
 3056   : 216   : verifycode:merge_registers
 2840   : 112   : verifycode:isAssignableTo
 2728   : 144   : verifycode:merge_fullinfo_types
 2584   : 128   : CVMjniGetSuperclass
 2456   : 120   : CVMjniNewLocalRef
 2336   : 144   : CVMjniCreateLocalRef
 2192   : 112   : CVMjniFatalError
 2080   : 120   : CVMdumpStack
 1960   : 128   : stacks:CVMdumpFrames
 1832   : 392   : CVMdumpFrame
 1440   : 120   : stacks:CVMdumpData
 1320   : 376   : CVMconsolePrintf
 944    : 304   : CVMformatStringVaList
 640    : 168   : CVMtypeidMethodTypeToCString
 472    : 112   : CVMtypeidFieldTypeToCString
 360    : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
 232    : 120   : typeid:conditionalPutstring
 112    : 112   : CVMCstrncpyStub
 0      : ?(0)   : strncpy
```

```
> stacktrace CVMclassLink .umul
 4408    : 128    : CVMclassLink
 4280    : 376    : CVMclassVerify
 3904    : 144    : VerifyClass
 3760    : 160    : verifycode:verify_method
 3600    : 184    : verifycode:run_dataflow
 3416    : 200    : verifycode:merge_into_successors
 3216    : 264    : verifycode:merge_into_one_successor
 2952    : 216    : verifycode:merge_registers
 2736    : 112    : verifycode:isAssignableTo
 2624    : 144    : verifycode:merge_fullinfo_types
 2480    : 128    : CVMjniGetSuperclass
 2352    : 120    : CVMjniNewLocalRef
 2232    : 144    : CVMjniCreateLocalRef
 2088    : 112    : CVMjniFatalError
 1976    : 120    : CVMdumpStack
 1856    : 128    : stacks:CVMdumpFrames
 1728    : 392    : CVMdumpFrame
 1336    : 120    : stacks:CVMdumpData
 1216    : 376    : CVMconsolePrintf
 840     : 304    : CVMformatStringVaList
 536     : 168    : CVMtypeidMethodTypeToCString
 368     : 128    : CVMtypeidGetSignatureIterator
 240     : 120    : CVMtypeidGetTerseSignatureIterator
 120     : 120    : typeid:indexSegmentedTable
 0       : ?(0)   : .umul

> stacktrace CVMclassLink strlen
 4392    : 128    : CVMclassLink
 4264    : 376    : CVMclassVerify
 3888    : 144    : VerifyClass
 3744    : 160    : verifycode:verify_method
 3584    : 184    : verifycode:run_dataflow
 3400    : 200    : verifycode:merge_into_successors
 3200    : 264    : verifycode:merge_into_one_successor
 2936    : 216    : verifycode:merge_registers
```

```
2720    : 112    : verifycode:isAssignableTo
2608    : 144    : verifycode:merge_fullinfo_types
2464    : 128    : CVMjniGetSuperclass
2336    : 120    : CVMjniNewLocalRef
2216    : 144    : CVMjniCreateLocalRef
2072    : 112    : CVMjniFatalError
1960    : 120    : CVMdumpStack
1840    : 128    : stacks:CVMdumpFrames
1712    : 392    : CVMdumpFrame
1320    : 120    : stacks:CVMdumpData
1200    : 376    : CVMconsolePrintf
824     : 304    : CVMformatStringVaList
520     : 168    : CVMtypeidMethodTypeToCString
352     : 112    : CVMtypeidFieldTypeToCString
240     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
112     : 112    : CVMCstrlenStub
0       : ?(0)   : strlen

> stacktrace CVMclassLink free
3984    : 128    : CVMclassLink
3856    : 376    : CVMclassVerify
3480    : 144    : VerifyClass
3336    : 160    : verifycode:verify_method
3176    : 184    : verifycode:run_dataflow
2992    : 200    : verifycode:merge_into_successors
2792    : 264    : verifycode:merge_into_one_successor
2528    : 216    : verifycode:merge_registers
2312    : 112    : verifycode:isAssignableTo
2200    : 144    : verifycode:merge_fullinfo_types
2056    : 128    : CVMjniGetSuperclass
1928    : 120    : CVMjniNewLocalRef
1808    : 144    : CVMjniCreateLocalRef
1664    : 112    : CVMjniFatalError
1552    : 120    : CVMdumpStack
1432    : 128    : stacks:CVMdumpFrames
1304    : 392    : CVMdumpFrame
```

```
 912    : 120   : stacks:CVMdumpData
 792    : 376   : CVMconsolePrintf
 416    : 304   : CVMformatStringVaList
 112    : 112   : CVMCfreeStub
 0      : ?(0)  : free


> stacktrace CVMclassLink  sprintf
 3872   : 128   : CVMclassLink
 3744   : 376   : CVMclassVerify
 3368   : 144   : VerifyClass
 3224   : 160   : verifycode:verify_method
 3064   : 184   : verifycode:run_dataflow
 2880   : 200   : verifycode:merge_into_successors
 2680   : 264   : verifycode:merge_into_one_successor
 2416   : 216   : verifycode:merge_registers
 2200   : 112   : verifycode:isAssignableTo
 2088   : 144   : verifycode:merge_fullinfo_types
 1944   : 128   : CVMjniGetSuperclass
 1816   : 120   : CVMjniNewLocalRef
 1696   : 144   : CVMjniCreateLocalRef
 1552   : 112   : CVMjniFatalError
 1440   : 120   : CVMdumpStack
 1320   : 128   : stacks:CVMdumpFrames
 1192   : 392   : CVMdumpFrame
 800    : 120   : stacks:CVMdumpData
 680    : 376   : CVMconsolePrintf
 304    : 304   : CVMformatStringVaList
 0      : ?(0)  : sprintf


> stacktrace CVMclassLink fflush
 3568   : 128   : CVMclassLink
 3440   : 376   : CVMclassVerify
 3064   : 144   : VerifyClass
 2920   : 160   : verifycode:verify_method
 2760   : 184   : verifycode:run_dataflow
 2576   : 200   : verifycode:merge_into_successors
```

```
2376    : 264    : verifycode:merge_into_one_successor
2112    : 216    : verifycode:merge_registers
1896    : 112    : verifycode:isAssignableTo
1784    : 144    : verifycode:merge_fullinfo_types
1640    : 128    : CVMjniGetSuperclass
1512    : 120    : CVMjniNewLocalRef
1392    : 144    : CVMjniCreateLocalRef
1248    : 112    : CVMjniFatalError
1136    : 120    : CVMdumpStack
1016    : 128    : stacks:CVMdumpFrames
888     : 392    : CVMdumpFrame
496     : 120    : stacks:CVMdumpData
376     : 376    : CVMconsolePrintf
0       : ?(0)   : fflush


> stacktrace CVMclassLink pthread_cond_wait
3504    : 128    : CVMclassLink
3376    : 376    : CVMclassVerify
3000    : 144    : VerifyClass
2856    : 160    : verifycode:verify_method
2696    : 184    : verifycode:run_dataflow
2512    : 200    : verifycode:merge_into_successors
2312    : 264    : verifycode:merge_into_one_successor
2048    : 216    : verifycode:merge_registers
1832    : 112    : verifycode:isAssignableTo
1720    : 144    : verifycode:merge_fullinfo_types
1576    : 112    :
verifycode:CVMCstackDummy2object_fullinfo_to_classclass
1464    : 136    : verifycode:object_fullinfo_to_classclass
1328    : 120    : CVMjniNewGlobalRef
1208    : 112    : CVMID_getGlobalRoot
1096    : 120    : globalroots:CVMIDprivate_getGlobalRoot
976     : 136    : globalroots:CVMIDprivate_getGlobalRootNoLock
840     : 136    : CVMexpandStack
704     : 136    : CVMcsRendezvous
568     : 128    : CVMsysMutexWait
```

```
440     : 128    : CVMreentrantMutexWait
312     : 144    : CVMcondvarWait
168     : 168    : POSIXcondvarWait
0       : ?(0)   : pthread_cond_wait


> stacktrace CVMclassLink pthread_mutex_unlock
3448    : 128    : CVMclassLink
3320    : 376    : CVMclassVerify
2944    : 144    : VerifyClass
2800    : 160    : verifycode:verify_method
2640    : 184    : verifycode:run_dataflow
2456    : 200    : verifycode:merge_into_successors
2256    : 264    : verifycode:merge_into_one_successor
1992    : 216    : verifycode:merge_registers
1776    : 112    : verifycode:isAssignableTo
1664    : 144    : verifycode:merge_fullinfo_types
1520    : 112    :
verifycode:CVMCstackDummy2object_fullinfo_to_classclass
1408    : 136    : verifycode:object_fullinfo_to_classclass
1272    : 120    : CVMjniNewGlobalRef
1152    : 112    : CVMID_getGlobalRoot
1040    : 120    : globalroots:CVMIDprivate_getGlobalRoot
920     : 136    : globalroots:CVMIDprivate_getGlobalRootNoLock
784     : 136    : CVMexpandStack
648     : 136    : CVMcsRendezvous
512     : 128    : CVMsysMutexWait
384     : 128    : CVMreentrantMutexWait
256     : 144    : CVMcondvarWait
112     : 112    : POSIXmutexUnlock
0       : ?(0)   : pthread_mutex_unlock
```

B.3.5 Worst C Stack Usage Chain in Formatting a String from the PC for a Console Print Out

Some worst cases of stack consumption occurring in the console print out routine that have a call path from CVMCstackDummy2CVMformatStringVaList to a C library function such as strncpy are given below.

```
> stacktrace utils:CVMCstackDummy2CVMformatStringVaList strncpy
  1792  : 112   : utils:CVMCstackDummy2CVMformatStringVaList
  1680  : 112   : CVMCstackCheckSize
  1568  : 136   : CVMcsRendezvous
  1432  : 112   : CVMsysMutexLock
  1320  : 376   : CVMconsolePrintf
   944  : 304   : CVMformatStringVaList
   640  : 168   : CVMtypeidMethodTypeToCString
   472  : 112   : CVMtypeidFieldTypeToCString
   360  : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
   232  : 120   : typeid:conditionalPutstring
   112  : 112   : CVMCstrncpyStub
     0  : ?(0)  : strncpy


> stacktrace utils:CVMCstackDummy2CVMformatStringVaList .umul
  1688  : 112   : utils:CVMCstackDummy2CVMformatStringVaList
  1576  : 112   : CVMCstackCheckSize
  1464  : 136   : CVMcsRendezvous
  1328  : 112   : CVMsysMutexLock
  1216  : 376   : CVMconsolePrintf
   840  : 304   : CVMformatStringVaList
   536  : 168   : CVMtypeidMethodTypeToCString
   368  : 128   : CVMtypeidGetSignatureIterator
   240  : 120   : CVMtypeidGetTerseSignatureIterator
   120  : 120   : typeid:indexSegmentedTable
     0  : ?(0)  : .umul


> stacktrace utils:CVMCstackDummy2CVMformatStringVaList strlen
  1672  : 112   : utils:CVMCstackDummy2CVMformatStringVaList
  1560  : 112   : CVMCstackCheckSize
```

```
1448    : 136    : CVMcsRendezvous
1312    : 112    : CVMsysMutexLock
1200    : 376    : CVMconsolePrintf
824     : 304    : CVMformatStringVaList
520     : 168    : CVMtypeidMethodTypeToCString
352     : 112    : CVMtypeidFieldTypeToCString
240     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
112     : 112    : CVMCstrlenStub
0       : ?(0)   : strlen

> stacktrace utils:CVMCstackDummy2CVMformatStringVaList
pthread_getspecific
1384    : 112    : utils:CVMCstackDummy2CVMformatStringVaList
1272    : 112    : CVMCstackCheckSize
1160    : 136    : CVMcsRendezvous
1024    : 112    : CVMsysMutexLock
912     : 376    : CVMconsolePrintf
536     : 304    : CVMformatStringVaList
232     : 120    : CVMgetEE
112     : 112    : POSIXthreadGetSelf
0       : ?(0)   : pthread_getspecific

> stacktrace utils:CVMCstackDummy2CVMformatStringVaList free
1264    : 112    : utils:CVMCstackDummy2CVMformatStringVaList
1152    : 112    : CVMCstackCheckSize
1040    : 136    : CVMcsRendezvous
904     : 112    : CVMsysMutexLock
792     : 376    : CVMconsolePrintf
416     : 304    : CVMformatStringVaList
112     : 112    : CVMCfreeStub
0       : ?(0)   : free

> stacktrace utils:CVMCstackDummy2CVMformatStringVaList sprintf
1152    : 112    : utils:CVMCstackDummy2CVMformatStringVaList
1040    : 112    : CVMCstackCheckSize
928     : 136    : CVMcsRendezvous
```

```
792     : 112    : CVMsysMutexLock
680     : 376    : CVMconsolePrintf
304     : 304    : CVMformatStringVaList
0       : ?(0)   : sprintf
```

B.3.6 Worst C Stack Usage Chain in Formatting a String of Object from a Class and Hash Value

Some worst cases of stack consumption occurring in the console print out routine which displays a string of objects from its class block or a hashed value have a call path from `CVMCstackDummy2CVMconsolePrintf` to a C library function such as `strncpy` are given below.

```
> stacktrace utils:CVMCstackDummy2CVMconsolePrintf strncpy
 1800    : 120    : utils:CVMCstackDummy2CVMconsolePrintf
 1680    : 112    : CVMCstackCheckSize
 1568    : 136    : CVMcsRendezvous
 1432    : 112    : CVMsysMutexLock
 1320    : 376    : CVMconsolePrintf
 944     : 304    : CVMformatStringVaList
 640     : 168    : CVMtypeidMethodTypeToCString
 472     : 112    : CVMtypeidFieldTypeToCString
 360     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
 232     : 120    : typeid:conditionalPutstring
 112     : 112    : CVMCstrncpyStub
 0       : ?(0)   : strncpy


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf .umul
 1696    : 120    : utils:CVMCstackDummy2CVMconsolePrintf
 1576    : 112    : CVMCstackCheckSize
 1464    : 136    : CVMcsRendezvous
 1328    : 112    : CVMsysMutexLock
 1216    : 376    : CVMconsolePrintf
 840     : 304    : CVMformatStringVaList
 536     : 168    : CVMtypeidMethodTypeToCString
 368     : 128    : CVMtypeidGetSignatureIterator
 240     : 120    : CVMtypeidGetTerseSignatureIterator
```

```
 120     : 120    : typeid:indexSegmentedTable
 0       : ?(0)   : .umul


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf strlen
 1680    : 120    : utils:CVMCstackDummy2CVMconsolePrintf
 1560    : 112    : CVMCstackCheckSize
 1448    : 136    : CVMcsRendezvous
 1312    : 112    : CVMsysMutexLock
 1200    : 376    : CVMconsolePrintf
 824     : 304    : CVMformatStringVaList
 520     : 168    : CVMtypeidMethodTypeToCString
 352     : 112    : CVMtypeidFieldTypeToCString
 240     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
 112     : 112    : CVMCstrlenStub
 0       : ?(0)   : strlen


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf
pthread_getspecific
 1392    : 120    : utils:CVMCstackDummy2CVMconsolePrintf
 1272    : 112    : CVMCstackCheckSize
 1160    : 136    : CVMcsRendezvous
 1024    : 112    : CVMsysMutexLock
 912     : 376    : CVMconsolePrintf
 536     : 304    : CVMformatStringVaList
 232     : 120    : CVMgetEE
 112     : 112    : POSIXthreadGetSelf
 0       : ?(0)   : pthread_getspecific


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf free
 1272    : 120    : utils:CVMCstackDummy2CVMconsolePrintf
 1152    : 112    : CVMCstackCheckSize
 1040    : 136    : CVMcsRendezvous
 904     : 112    : CVMsysMutexLock
 792     : 376    : CVMconsolePrintf
 416     : 304    : CVMformatStringVaList
 112     : 112    : CVMCfreeStub
```

```
0      : ?(0)  : free


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf sprintf
1160   : 120   : utils:CVMCstackDummy2CVMconsolePrintf
1040   : 112   : CVMCstackCheckSize
928    : 136   : CVMcsRendezvous
792    : 112   : CVMsysMutexLock
680    : 376   : CVMconsolePrintf
304    : 304   : CVMformatStringVaList
0      : ?(0)  : sprintf


> stacktrace utils:CVMCstackDummy2CVMconsolePrintf
pthread_cond_wait
936    : 120   : utils:CVMCstackDummy2CVMconsolePrintf
816    : 112   : CVMCstackCheckSize
704    : 136   : CVMcsRendezvous
568    : 128   : CVMsysMutexWait
440    : 128   : CVMreentrantMutexWait
312    : 144   : CVMcondvarWait
168    : 168   : POSIXcondvarWait
0      : ?(0)  : pthread_cond_wait


> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe strncpy
2056   : 112   : utils:CVMCstackDummy2CVMobjectGetHashSafe
1944   : 120   : CVMobjectGetHashSafe
1824   : 120   : objsync:CVMobjectComputeHash
1704   : 136   : objsync:CVMobjectSetHashCode
1568   : 136   : CVMcsRendezvous
1432   : 112   : CVMsysMutexLock
1320   : 376   : CVMconsolePrintf
944    : 304   : CVMformatStringVaList
640    : 168   : CVMtypeidMethodTypeToCString
472    : 112   : CVMtypeidFieldTypeToCString
360    : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
232    : 120   : typeid:conditionalPutstring
112    : 112   : CVMCstrncpyStub
```

```
 0       : ?(0)   : strncpy


> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe .umul
 1952    : 112    : utils:CVMCstackDummy2CVMobjectGetHashSafe
 1840    : 120    : CVMobjectGetHashSafe
 1720    : 120    : objsync:CVMobjectComputeHash
 1600    : 136    : objsync:CVMobjectSetHashCode
 1464    : 136    : CVMcsRendezvous
 1328    : 112    : CVMsysMutexLock
 1216    : 376    : CVMconsolePrintf
 840     : 304    : CVMformatStringVaList
 536     : 168    : CVMtypeidMethodTypeToCString
 368     : 128    : CVMtypeidGetSignatureIterator
 240     : 120    : CVMtypeidGetTerseSignatureIterator
 120     : 120    : typeid:indexSegmentedTable
 0       : ?(0)   : .umul


> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe strlen
 1936    : 112    : utils:CVMCstackDummy2CVMobjectGetHashSafe
 1824    : 120    : CVMobjectGetHashSafe
 1704    : 120    : objsync:CVMobjectComputeHash
 1584    : 136    : objsync:CVMobjectSetHashCode
 1448    : 136    : CVMcsRendezvous
 1312    : 112    : CVMsysMutexLock
 1200    : 376    : CVMconsolePrintf
 824     : 304    : CVMformatStringVaList
 520     : 168    : CVMtypeidMethodTypeToCString
 352     : 112    : CVMtypeidFieldTypeToCString
 240     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
 112     : 112    : CVMCstrlenStub
 0       : ?(0)   : strlen


> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe
pthread_getspecific
 1648    : 112    : utils:CVMCstackDummy2CVMobjectGetHashSafe
 1536    : 120    : CVMobjectGetHashSafe
```

```
 1416    : 120   : objsync:CVMobjectComputeHash
 1296    : 136   : objsync:CVMobjectSetHashCode
 1160    : 136   : CVMcsRendezvous
 1024    : 112   : CVMsysMutexLock
 912     : 376   : CVMconsolePrintf
 536     : 304   : CVMformatStringVaList
 232     : 120   : CVMgetEE
 112     : 112   : POSIXthreadGetSelf
 0       : ?(0)  : pthread_getspecific

> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe free
 1528    : 112   : utils:CVMCstackDummy2CVMobjectGetHashSafe
 1416    : 120   : CVMobjectGetHashSafe
 1296    : 120   : objsync:CVMobjectComputeHash
 1176    : 136   : objsync:CVMobjectSetHashCode
 1040    : 136   : CVMcsRendezvous
 904     : 112   : CVMsysMutexLock
 792     : 376   : CVMconsolePrintf
 416     : 304   : CVMformatStringVaList
 112     : 112   : CVMCfreeStub
 0       : ?(0)  : free

> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe sprintf
 1416    : 112   : utils:CVMCstackDummy2CVMobjectGetHashSafe
 1304    : 120   : CVMobjectGetHashSafe
 1184    : 120   : objsync:CVMobjectComputeHash
 1064    : 136   : objsync:CVMobjectSetHashCode
 928     : 136   : CVMcsRendezvous
 792     : 112   : CVMsysMutexLock
 680     : 376   : CVMconsolePrintf
 304     : 304   : CVMformatStringVaList
 0       : ?(0)  : sprintf

> stacktrace utils:CVMCstackDummy2CVMobjectGetHashSafe
pthread_cond_wait
 1192    : 112   : utils:CVMCstackDummy2CVMobjectGetHashSafe
```

```
1080    : 120    : CVMobjectGetHashSafe
960     : 120    : objsync:CVMobjectComputeHash
840     : 136    : objsync:CVMobjectSetHashCode
704     : 136    : CVMcsRendezvous
568     : 128    : CVMsysMutexWait
440     : 128    : CVMreentrantMutexWait
312     : 144    : CVMcondvarWait
168     : 168    : POSIXcondvarWait
0       : ?(0)   : pthread_cond_wait
```

B.3.7   Worst C Stack Usage Chain in Class Verifier

Some worst cases of stack consumption occurring in class verifier that merges types
of two array objects have a call path from
`CVMCstackDummy2merge_fullinfo_types` to a C library function such as
`strncpy`.

```
> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types strncpy
 2848    : 120    : verifycode:CVMCstackDummy2merge_fullinfo_types
 2728    : 144    : verifycode:merge_fullinfo_types
 2584    : 128    : CVMjniGetSuperclass
 2456    : 120    : CVMjniNewLocalRef
 2336    : 144    : CVMjniCreateLocalRef
 2192    : 112    : CVMjniFatalError
 2080    : 120    : CVMdumpStack
 1960    : 128    : stacks:CVMdumpFrames
 1832    : 392    : CVMdumpFrame
 1440    : 120    : stacks:CVMdumpData
 1320    : 376    : CVMconsolePrintf
 944     : 304    : CVMformatStringVaList
 640     : 168    : CVMtypeidMethodTypeToCString
 472     : 112    : CVMtypeidFieldTypeToCString
 360     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
 232     : 120    : typeid:conditionalPutstring
 112     : 112    : CVMCstrncpyStub
 0       : ?(0)   : strncpy
```

```
> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types .umul
 2744    : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 2624    : 144   : verifycode:merge_fullinfo_types
 2480    : 128   : CVMjniGetSuperclass
 2352    : 120   : CVMjniNewLocalRef
 2232    : 144   : CVMjniCreateLocalRef
 2088    : 112   : CVMjniFatalError
 1976    : 120   : CVMdumpStack
 1856    : 128   : stacks:CVMdumpFrames
 1728    : 392   : CVMdumpFrame
 1336    : 120   : stacks:CVMdumpData
 1216    : 376   : CVMconsolePrintf
 840     : 304   : CVMformatStringVaList
 536     : 168   : CVMtypeidMethodTypeToCString
 368     : 128   : CVMtypeidGetSignatureIterator
 240     : 120   : CVMtypeidGetTerseSignatureIterator
 120     : 120   : typeid:indexSegmentedTable
 0       : ?(0)  : .umul

> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types strlen
 2728    : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 2608    : 144   : verifycode:merge_fullinfo_types
 2464    : 128   : CVMjniGetSuperclass
 2336    : 120   : CVMjniNewLocalRef
 2216    : 144   : CVMjniCreateLocalRef
 2072    : 112   : CVMjniFatalError
 1960    : 120   : CVMdumpStack
 1840    : 128   : stacks:CVMdumpFrames
 1712    : 392   : CVMdumpFrame
 1320    : 120   : stacks:CVMdumpData
 1200    : 376   : CVMconsolePrintf
 824     : 304   : CVMformatStringVaList
 520     : 168   : CVMtypeidMethodTypeToCString
 352     : 112   : CVMtypeidFieldTypeToCString
 240     : 128   : typeid:CVMtypeidPrivateFieldTypeToCString
 112     : 112   : CVMCstrlenStub
```

```
    0      : ?(0)  : strlen


> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types
pthread_getspecific
 2440   : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 2320   : 144   : verifycode:merge_fullinfo_types
 2176   : 128   : CVMjniGetSuperclass
 2048   : 120   : CVMjniNewLocalRef
 1928   : 144   : CVMjniCreateLocalRef
 1784   : 112   : CVMjniFatalError
 1672   : 120   : CVMdumpStack
 1552   : 128   : stacks:CVMdumpFrames
 1424   : 392   : CVMdumpFrame
 1032   : 120   : stacks:CVMdumpData
 912    : 376   : CVMconsolePrintf
 536    : 304   : CVMformatStringVaList
 232    : 120   : CVMgetEE
 112    : 112   : POSIXthreadGetSelf
 0      : ?(0)  : pthread_getspecific


> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types  free
 2320   : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 2200   : 144   : verifycode:merge_fullinfo_types
 2056   : 128   : CVMjniGetSuperclass
 1928   : 120   : CVMjniNewLocalRef
 1808   : 144   : CVMjniCreateLocalRef
 1664   : 112   : CVMjniFatalError
 1552   : 120   : CVMdumpStack
 1432   : 128   : stacks:CVMdumpFrames
 1304   : 392   : CVMdumpFrame
 912    : 120   : stacks:CVMdumpData
 792    : 376   : CVMconsolePrintf
 416    : 304   : CVMformatStringVaList
 112    : 112   : CVMCfreeStub
 0      : ?(0)  : free
```

```
> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types sprintf
 2208   : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 2088   : 144   : verifycode:merge_fullinfo_types
 1944   : 128   : CVMjniGetSuperclass
 1816   : 120   : CVMjniNewLocalRef
 1696   : 144   : CVMjniCreateLocalRef
 1552   : 112   : CVMjniFatalError
 1440   : 120   : CVMdumpStack
 1320   : 128   : stacks:CVMdumpFrames
 1192   : 392   : CVMdumpFrame
 800    : 120   : stacks:CVMdumpData
 680    : 376   : CVMconsolePrintf
 304    : 304   : CVMformatStringVaList
 0      : ?(0)  : sprintf

> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types  fflush
 1904   : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 1784   : 144   : verifycode:merge_fullinfo_types
 1640   : 128   : CVMjniGetSuperclass
 1512   : 120   : CVMjniNewLocalRef
 1392   : 144   : CVMjniCreateLocalRef
 1248   : 112   : CVMjniFatalError
 1136   : 120   : CVMdumpStack
 1016   : 128   : stacks:CVMdumpFrames
 888    : 392   : CVMdumpFrame
 496    : 120   : stacks:CVMdumpData
 376    : 376   : CVMconsolePrintf
 0      : ?(0)  : fflush

> stacktrace verifycode:CVMCstackDummy2merge_fullinfo_types
pthread_cond_
wait
 1840   : 120   : verifycode:CVMCstackDummy2merge_fullinfo_types
 1720   : 144   : verifycode:merge_fullinfo_types
 1576   : 112   :
verifycode:CVMCstackDummy2object_fullinfo_to_classclass
```

```
1464   : 136   : verifycode:object_fullinfo_to_classclass
1328   : 120   : CVMjniNewGlobalRef
1208   : 112   : CVMID_getGlobalRoot
1096   : 120   : globalroots:CVMIDprivate_getGlobalRoot
976    : 136   : globalroots:CVMIDprivate_getGlobalRootNoLock
840    : 136   : CVMexpandStack
704    : 136   : CVMcsRendezvous
568    : 128   : CVMsysMutexWait
440    : 128   : CVMreentrantMutexWait
312    : 144   : CVMcondvarWait
168    : 168   : POSIXcondvarWait
0      : ?(0)  : pthread_cond_wait
```

B.3.8 Worst C Stack Usage Chain in Signaling an Exception Routine

Some worst cases of stack consumption occurring in a function that signals a thrown exception have a call path from `CVMsignalErrorVaList` to a C library function such as `strncpy`.

```
> stacktrace CVMsignalErrorVaList strncpy
 5448   : 120   : CVMsignalErrorVaList
 5328   : 144   : interpreter:CVMgcSafeSignalError
 5184   : 112   : CVMfillInStackTrace
 5072   : 200   : jvm:CVMgcUnsafeFillInStackTrace
 4872   : 376   : CVMnewStringUTF
 4496   : 144   : CVMnewString
 4352   : 136   : CVMgcAllocNewInstance
 4216   : 128   : CVMjniCallStaticVoidMethod
 4088   : 184   : jni_impl:CVMjniInvoke
 3904   : 136   : CVMjniExceptionDescribe
 3768   : 112   : CVMjniGetMethodID
 3656   : 136   : jni_impl:CVMjniGetXMethodID
 3520   : 112   : CVMclassInit

 3408   : 128   : classinitialize:CVMprivateClassInit
 3280   : 128   : CVMjniCallVoidMethod
```

```
3152  : 184  : jni_impl:CVMjniInvoke
2968  : 136  : CVMjniExceptionDescribe

2832  : 112  : CVMjniGetMethodID
2720  : 136  : jni_impl:CVMjniGetXMethodID
2584  : 128  : CVMtypeidNewMethodIDFromNameAndSig
2456  : 240  : typeid:referenceMethodSignature
2216  : 144  : typeid:referenceFieldSignature
2072  : 120  : typeid:referenceClassName
1952  : 144  : typeid:lookupClass
1808  : 120  : typeid:findFreeScalarEntry
1688  : 136  : typeid:findFreeTableEntry
1552  : 120  : typeid:unlockThrowOutOfMemoryError
1432  : 112  : CVMsysMutexUnlock
1320  : 376  : CVMconsolePrintf
944   : 304  : CVMformatStringVaList
640   : 168  : CVMtypeidMethodTypeToCString
472   : 112  : CVMtypeidFieldTypeToCString
360   : 128  : typeid:CVMtypeidPrivateFieldTypeToCString
232   : 120  : typeid:conditionalPutstring
112   : 112  : CVMCstrncpyStub
0     : ?(0) : strncpy

> stacktrace CVMsignalErrorVaList .umul
5344  : 120  : CVMsignalErrorVaList
5224  : 144  : interpreter:CVMgcSafeSignalError
5080  : 112  : CVMfillInStackTrace
4968  : 200  : jvm:CVMgcUnsafeFillInStackTrace
4768  : 376  : CVMnewStringUTF
4392  : 144  : CVMnewString
4248  : 136  : CVMgcAllocNewInstance
4112  : 128  : CVMjniCallStaticVoidMethod
3984  : 184  : jni_impl:CVMjniInvoke
3800  : 136  : CVMjniExceptionDescribe
3664  : 112  : CVMjniGetMethodID
3552  : 136  : jni_impl:CVMjniGetXMethodID
```

```
3416  : 112  : CVMclassInit
3304  : 128  : classinitialize:CVMprivateClassInit
3176  : 128  : CVMjniCallVoidMethod
3048  : 184  : jni_impl:CVMjniInvoke
2864  : 136  : CVMjniExceptionDescribe
2728  : 112  : CVMjniGetMethodID
2616  : 136  : jni_impl:CVMjniGetXMethodID
2480  : 128  : CVMtypeidNewMethodIDFromNameAndSig
2352  : 240  : typeid:referenceMethodSignature
2112  : 144  : typeid:referenceFieldSignature
1968  : 120  : typeid:referenceClassName
1848  : 144  : typeid:lookupClass
1704  : 120  : typeid:findFreeScalarEntry
1584  : 136  : typeid:findFreeTableEntry
1448  : 120  : typeid:unlockThrowOutOfMemoryError
1328  : 112  : CVMsysMutexUnlock
1216  : 376  : CVMconsolePrintf
840   : 304  : CVMformatStringVaList
536   : 168  : CVMtypeidMethodTypeToCString
368   : 128  : CVMtypeidGetSignatureIterator
240   : 120  : CVMtypeidGetTerseSignatureIterator
120   : 120  : typeid:indexSegmentedTable
0     : ?(0) : .umul

> stacktrace CVMsignalErrorVaList strlen
5328  : 120  : CVMsignalErrorVaList
5208  : 144  : interpreter:CVMgcSafeSignalError
5064  : 112  : CVMfillInStackTrace
4952  : 200  : jvm:CVMgcUnsafeFillInStackTrace
4752  : 376  : CVMnewStringUTF
4376  : 144  : CVMnewString
4232  : 136  : CVMgcAllocNewInstance
4096  : 128  : CVMjniCallStaticVoidMethod
3968  : 184  : jni_impl:CVMjniInvoke
3784  : 136  : CVMjniExceptionDescribe
3648  : 112  : CVMjniGetMethodID
```

```
3536    : 136    : jni_impl:CVMjniGetXMethodID
3400    : 112    : CVMclassInit
3288    : 128    : classinitialize:CVMprivateClassInit
3160    : 128    : CVMjniCallVoidMethod
3032    : 184    : jni_impl:CVMjniInvoke
2848    : 136    : CVMjniExceptionDescribe
2712    : 112    : CVMjniGetMethodID
2600    : 136    : jni_impl:CVMjniGetXMethodID
2464    : 128    : CVMtypeidNewMethodIDFromNameAndSig
2336    : 240    : typeid:referenceMethodSignature
2096    : 144    : typeid:referenceFieldSignature
1952    : 120    : typeid:referenceClassName
1832    : 144    : typeid:lookupClass
1688    : 120    : typeid:findFreeScalarEntry
1568    : 136    : typeid:findFreeTableEntry
1432    : 120    : typeid:unlockThrowOutOfMemoryError
1312    : 112    : CVMsysMutexUnlock
1200    : 376    : CVMconsolePrintf
824     : 304    : CVMformatStringVaList
520     : 168    : CVMtypeidMethodTypeToCString
352     : 112    : CVMtypeidFieldTypeToCString
240     : 128    : typeid:CVMtypeidPrivateFieldTypeToCString
112     : 112    : CVMCstrlenStub
0       : ?(0)   : strlen

> stacktrace CVMsignalErrorVaList pthread_getspecific
 5040    : 120    : CVMsignalErrorVaList
 4920    : 144    : interpreter:CVMgcSafeSignalError
 4776    : 112    : CVMfillInStackTrace
 4664    : 200    : jvm:CVMgcUnsafeFillInStackTrace
 4464    : 376    : CVMnewStringUTF
 4088    : 144    : CVMnewString
 3944    : 136    : CVMgcAllocNewInstance
 3808    : 128    : CVMjniCallStaticVoidMethod
 3680    : 184    : jni_impl:CVMjniInvoke
 3496    : 136    : CVMjniExceptionDescribe
```

```
3360  : 112   : CVMjniGetMethodID
3248  : 136   : jni_impl:CVMjniGetXMethodID
3112  : 112   : CVMclassInit
3000  : 128   : classinitialize:CVMprivateClassInit
2872  : 128   : CVMjniCallVoidMethod
2744  : 184   : jni_impl:CVMjniInvoke
2560  : 136   : CVMjniExceptionDescribe
2424  : 112   : CVMjniGetMethodID
2312  : 136   : jni_impl:CVMjniGetXMethodID
2176  : 128   : CVMtypeidNewMethodIDFromNameAndSig
2048  : 240   : typeid:referenceMethodSignature
1808  : 144   : typeid:referenceFieldSignature
1664  : 120   : typeid:referenceClassName
1544  : 144   : typeid:lookupClass
1400  : 120   : typeid:findFreeScalarEntry
1280  : 136   : typeid:findFreeTableEntry
1144  : 120   : typeid:unlockThrowOutOfMemoryError
1024  : 112   : CVMsysMutexUnlock
912   : 376   : CVMconsolePrintf
536   : 304   : CVMformatStringVaList
232   : 120   : CVMgetEE
112   : 112   : POSIXthreadGetSelf
0     : ?(0)  : pthread_getspecific

> stacktrace CVMsignalErrorVaList free
4920  : 120   : CVMsignalErrorVaList
4800  : 144   : interpreter:CVMgcSafeSignalError
4656  : 112   : CVMfillInStackTrace
4544  : 200   : jvm:CVMgcUnsafeFillInStackTrace
4344  : 376   : CVMnewStringUTF
3968  : 144   : CVMnewString
3824  : 136   : CVMgcAllocNewInstance
3688  : 128   : CVMjniCallStaticVoidMethod
3560  : 184   : jni_impl:CVMjniInvoke
3376  : 136   : CVMjniExceptionDescribe
3240  : 112   : CVMjniGetMethodID
```

```
3128    : 136    : jni_impl:CVMjniGetXMethodID
2992    : 112    : CVMclassInit
2880    : 128    : classinitialize:CVMprivateClassInit
2752    : 128    : CVMjniCallVoidMethod
2624    : 184    : jni_impl:CVMjniInvoke
2440    : 136    : CVMjniExceptionDescribe
2304    : 112    : CVMjniGetMethodID
2192    : 136    : jni_impl:CVMjniGetXMethodID
2056    : 128    : CVMtypeidNewMethodIDFromNameAndSig
1928    : 240    : typeid:referenceMethodSignature
1688    : 144    : typeid:referenceFieldSignature
1544    : 120    : typeid:referenceClassName
1424    : 144    : typeid:lookupClass
1280    : 120    : typeid:findFreeScalarEntry
1160    : 136    : typeid:findFreeTableEntry
1024    : 120    : typeid:unlockThrowOutOfMemoryError
904     : 112    : CVMsysMutexUnlock
792     : 376    : CVMconsolePrintf
416     : 304    : CVMformatStringVaList
112     : 112    : CVMCfreeStub
0       : ?(0)   : free

> stacktrace CVMsignalErrorVaList sprintf
4808    : 120    : CVMsignalErrorVaList
4688    : 144    : interpreter:CVMgcSafeSignalError
4544    : 112    : CVMfillInStackTrace
4432    : 200    : jvm:CVMgcUnsafeFillInStackTrace
4232    : 376    : CVMnewStringUTF
3856    : 144    : CVMnewString
3712    : 136    : CVMgcAllocNewInstance
3576    : 128    : CVMjniCallStaticVoidMethod
3448    : 184    : jni_impl:CVMjniInvoke
3264    : 136    : CVMjniExceptionDescribe
3128    : 112    : CVMjniGetMethodID
3016    : 136    : jni_impl:CVMjniGetXMethodID
2880    : 112    : CVMclassInit
```

```
2768   : 128   : classinitialize:CVMprivateClassInit
2640   : 128   : CVMjniCallVoidMethod
2512   : 184   : jni_impl:CVMjniInvoke
2328   : 136   : CVMjniExceptionDescribe
2192   : 112   : CVMjniGetMethodID
2080   : 136   : jni_impl:CVMjniGetXMethodID
1944   : 128   : CVMtypeidNewMethodIDFromNameAndSig
1816   : 240   : typeid:referenceMethodSignature
1576   : 144   : typeid:referenceFieldSignature
1432   : 120   : typeid:referenceClassName
1312   : 144   : typeid:lookupClass
1168   : 120   : typeid:findFreeScalarEntry
1048   : 136   : typeid:findFreeTableEntry
912    : 120   : typeid:unlockThrowOutOfMemoryError
792    : 112   : CVMsysMutexUnlock
680    : 376   : CVMconsolePrintf
304    : 304   : CVMformatStringVaList
0      : ?(0)  : sprintf

> stacktrace CVMsignalErrorVaList fflush
 4504   : 120   : CVMsignalErrorVaList
 4384   : 144   : interpreter:CVMgcSafeSignalError
 4240   : 112   : CVMfillInStackTrace
 4128   : 200   : jvm:CVMgcUnsafeFillInStackTrace
 3928   : 376   : CVMnewStringUTF
 3552   : 144   : CVMnewString
 3408   : 136   : CVMgcAllocNewInstance
 3272   : 128   : CVMjniCallStaticVoidMethod
 3144   : 184   : jni_impl:CVMjniInvoke
 2960   : 136   : CVMjniExceptionDescribe
 2824   : 112   : CVMjniGetMethodID
 2712   : 136   : jni_impl:CVMjniGetXMethodID
 2576   : 112   : CVMclassInit
 2464   : 128   : classinitialize:CVMprivateClassInit
 2336   : 128   : CVMjniCallVoidMethod
 2208   : 184   : jni_impl:CVMjniInvoke
```

```
2024  : 136  : CVMjniExceptionDescribe
1888  : 112  : CVMjniGetMethodID
1776  : 136  : jni_impl:CVMjniGetXMethodID
1640  : 128  : CVMtypeidNewMethodIDFromNameAndSig
1512  : 240  : typeid:referenceMethodSignature
1272  : 144  : typeid:referenceFieldSignature
1128  : 120  : typeid:referenceClassName
1008  : 144  : typeid:lookupClass
864   : 120  : typeid:findFreeScalarEntry
744   : 136  : typeid:findFreeTableEntry
608   : 120  : typeid:unlockThrowOutOfMemoryError
488   : 112  : CVMsysMutexUnlock
376   : 376  : CVMconsolePrintf
0     : ?(0) : fflush
```