CDC Build System Guide

Java™ Platform, Micro Edition

Connected Device Configuration, Version 1.1.2

Foundation Profile, Version 1.1.2

Optimized Implementation

Sun Microsystems, Inc.
www.sun.com

December 2008

Contents

Figures

Tables

Preface

This guide describes the build system shared by various implementations of technology based on the Connected Device Configuration (CDC) and its related profiles and optional packages. The CDC build system can generate an executable binary image containing a CDC Java runtime environment.

This guide contains task descriptions for installing, configuring, testing and using the CDC build system as well as build option descriptions for controlling functionality, testability and performance features.

The companion document *CDC Runtime Guide* describes how to use a CDC Java runtime environment. It focuses on runtime issues like installation, configuration, testing and running Java™ technology-based application software as well as developer issues like compiling, debugging and profiling. This guide focuses on how to enable these features at build-time.

Who Should Read This Guide

This guide is intended for software engineers who need to build a CDC Java runtime environment for one of the following purposes:

- Porting the CDC HotSpot Implementation Java virtual machine.
- Porting one of the CDC profile class libraries.
- Testing a CDC Java runtime environment.
- Developing applications.
- System integration.

The reader should be familiar with Java and UNIX build tools as well as embedded software development. Before using the CDC build system, it is helpful to spend some time learning how to use a CDC Java runtime environment. See the *CDC Runtime Guide* for more details.

How This Book Is Organized

- Chapter 1 describes the concepts behind the CDC build system.
- Chapter 2 describes how to install and configure the CDC build system.
- Chapter 3 describes the organization of the CDC build system.
- Chapter 4 describes how to use the CDC build system to perform a test build and create a runtime environment deployment bundle.
- Appendix A describes the configuration options for the CDC build system.
- Appendix B describes the `JavaCodeCompact` build tool that is used for preloading system and application classes.
- Appendix C describes how to enable legacy profiling support for a CDC Java runtime environment.

CDC Software Releases

CDC technology is delivered by Sun through different kinds of software releases. The following technology releases are relevant to this guide:

- A *reference implementation* (RI) demonstrates CDC technology. CDC RIs are based on a common desktop development environment like Suse Linux 9.1.
- An *optimized implementation* (OI) supports strategic platforms and provide the basis for porting projects. The supported optimized implementation is based on the Linux platform and several embedded processors, including ARM and MIPS. Starter ports for other OS/CPU combinations are available from Java Partner Engineering (JPE).

This build guide describes the build system common to both of these source releases.

phoneME Open Source Project

Sun makes Java ME technology available through both a commercial license and the open source phoneME project (`https://phoneme.dev.java.net`). The main differences between the commercial and open source versions are:

- The commercial version is a superset of the open source version and contains additional security features that cannot be made available in source form as well as third-party components that may have restrictions on redistribution.

- The commercial version has had more rigorous software testing.
- The open source version represents active engineering development and so may have new features that have not been tested to the level that the commercial version requires.

The phoneME project includes several subprojects including *phoneME Advanced*, which corresponds with CDC technology and *phoneME Feature*, which corresponds with CLDC technology. See the phoneME Advanced Twiki at `http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced` for the latest information about the phoneME Advanced open source project.

Accessing Sun Resources Online

Sun provides online documentation resources for developers and licensees.

TABLE P-1 Sun Documentation Resources

URL	Description
`http://docs.sun.com`	Sun product documentation
`http://java.sun.com/javame/reference/index.jsp`	Java ME technical documentation
`http://developer.java.sun.com`	Java Developer Services
`https://java-partner.sun.com`	Java Partner Engineering
`http://java.net`	An open community that facilitates Java technology collaboration.
`http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced`	phoneME Advanced Twiki

Related Documentation

TABLE P-2 Related Documentation

Title	Description
CDC Runtime Guide	Runtime-oriented information for developers and testers.
CDC Porting Guide	Procedures and interface definitions for porting the CDC Java virtual machine and class library to an alternate target platform.
• *CDC Technology Compatibility Kit User's Guide* • *Foundation Profile Technology Compatibility Kit User's Guide* • *Security Optional Package Technology Compatibility Kit User's Guide*	User documentation for running the TCK validation suites.
Java Virtual Machine Specification, Second Edition	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: Programmer's Guide and Specification	Describes the native method interface used by the CDC HotSpot Implementation Java virtual machine. `http://java.sun.com/docs/books/jni`.
Java Virtual Machine Tools Interface (JVMTI)	Defines an interface that allows developer tools like `jdb` and third-party debuggers to interact with a debugger-capable Java runtime environment. See `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html`.
Inside Java 2 Platform Security	Describes the Java security framework, including security architecture, deployment and customization. See `http://java.sun.com/docs/books/security`.

Typographic Conventions

TABLE P-3 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*.

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 programming 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 based on 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-comments@java.sun.com`

You can send comments and suggestions regarding this guide by sending email to:

`docs@java.sun.com`

CHAPTER 1

Introduction

The CDC build system is a set of makefiles, scripts and tools that constructs a CDC Java runtime environment. The CDC build system uses commonly available Java and UNIX build tools to compile Java, C and assembly language source code and generate an executable image for a specific target platform. Build options control features of the generated CDC Java runtime environment that range from debugging capabilities and performance characteristics to optional functionality.

This chapter introduces the concepts and procedures of the CDC build system. The CDC build system has several purposes:

- Building different implementations of the CDC Java runtime environment for target platforms.
- Supporting application and runtime development.

The CDC build system can also be adapted to support different purposes. It can be:

- *Configured* to enable/disable functionality, testing and performance features.
- *Extended* to support optional packages, applications and class libraries.
- *Ported* to support new target platforms and devices.

Once built, the CDC Java runtime environment can be used in several different contexts:

- Runtime testing.
- TCK verification.
- Application development and testing.
- Product deployment.

The CDC build system operates on several host development platforms, including Solaris and Linux. The CDC build system uses cross-compilation to generate an executable image that can be transferred to a target platform for testing or deployment. For example, the CDC Java runtime environment can be built on a Linux-based x86/PC and then run on a Linux-based test device with an embedded RISC CPU like ARM or MIPS.

1.1 Host Development Environment

The CDC build system is based on commonly available software development tools. These include both Java development tools like `javac`, the Java compiler and UNIX development tools like `gcc`, `make` and `lex`. The UNIX development tools are further divided between *host build tools* that generate objects and resources for use within the CDC build system and *target build tools* that generate objects and resources for the target platform.

FIGURE 1-1 describes the basic workflow of the CDC build system and how it constructs a CDC Java runtime environment for a target platform. Chapter 2 describes the system requirements for the CDC build system.

1.2 Target Platforms

The implementations of the CDC Java runtime environment described in this guide are based on the Linux platform and ARM and MIPS CPUs. For help with starter ports that support other operating systems and CPUs, contact Java Partner Engineering (`http://www.sun.com/software/jpe`).

1.3 Build Options

The CDC build system has a variety of build options described in Appendix A that control different features of the CDC Java runtime environment:

- *Target devices.* At the top-level, the CDC build system supports several different target devices like PDAs or network routers.
- *Performance and testability.* Some build options control developer features or performance tradeoffs.
- *Standard API Features.* Java ME standards provide a flexible mechanism for constructing different yet conforming versions of a Java runtime environment. API choices balance the needs of product designers and application developers. Product designers can select standard API features that match the capabilities of their devices while application developers can use standard APIs shared by a range of different target devices.

1.4 Java ME Standard API Choices

The standard API choices available in Java ME technology are based on configurations, profiles and optional packages described in TABLE 1-1. To construct a conforming CDC Java runtime environment, a product designer chooses a configuration, a profile and any number of optional packages.

TABLE 1-1 Standard Java ME API Features

Configuration	Profile	Optional Package
CDC	Foundation	RMI
	Personal Basis	JDBC
	Personal	Security

For example, a product designer could choose the Connected Device Configuration (CDC), the Personal Profile and the RMI and JDBC optional packages. See Section 4.1.3, "Standard API Choices: Profiles and Optional Packages" on page 4-2 for information about how to use build options to select standard API features. See Section 4.3, "Building a Development Version of the CDC Java Class Library" on page 4-6 for information about how to build a target development version of the CDC Java class library.

1.5 Application Development

The CDC application developer compiles Java source code against a CDC Java class library and then runs the compiled application with a CDC Java runtime environment for testing and debugging. The companion document *CDC Runtime Guide* describes how to compile, run, debug and profile Java applications for the CDC platform using conventional Java SE tools and the CDC Java runtime environment.

In general, CDC application development is separate from runtime development. But there is one scenario where they cross paths. The CDC build system can be used as part of an application development workflow to bundle Java applications directly into a CDC Java runtime environment for the purposes of both performance and convenience. This capability is based on the preloading mechanism described in Section 4.4, "Preloading Java Class Files with `JavaCodeCompact`" on page 4-7.

1.6 CDC Build Process Overview

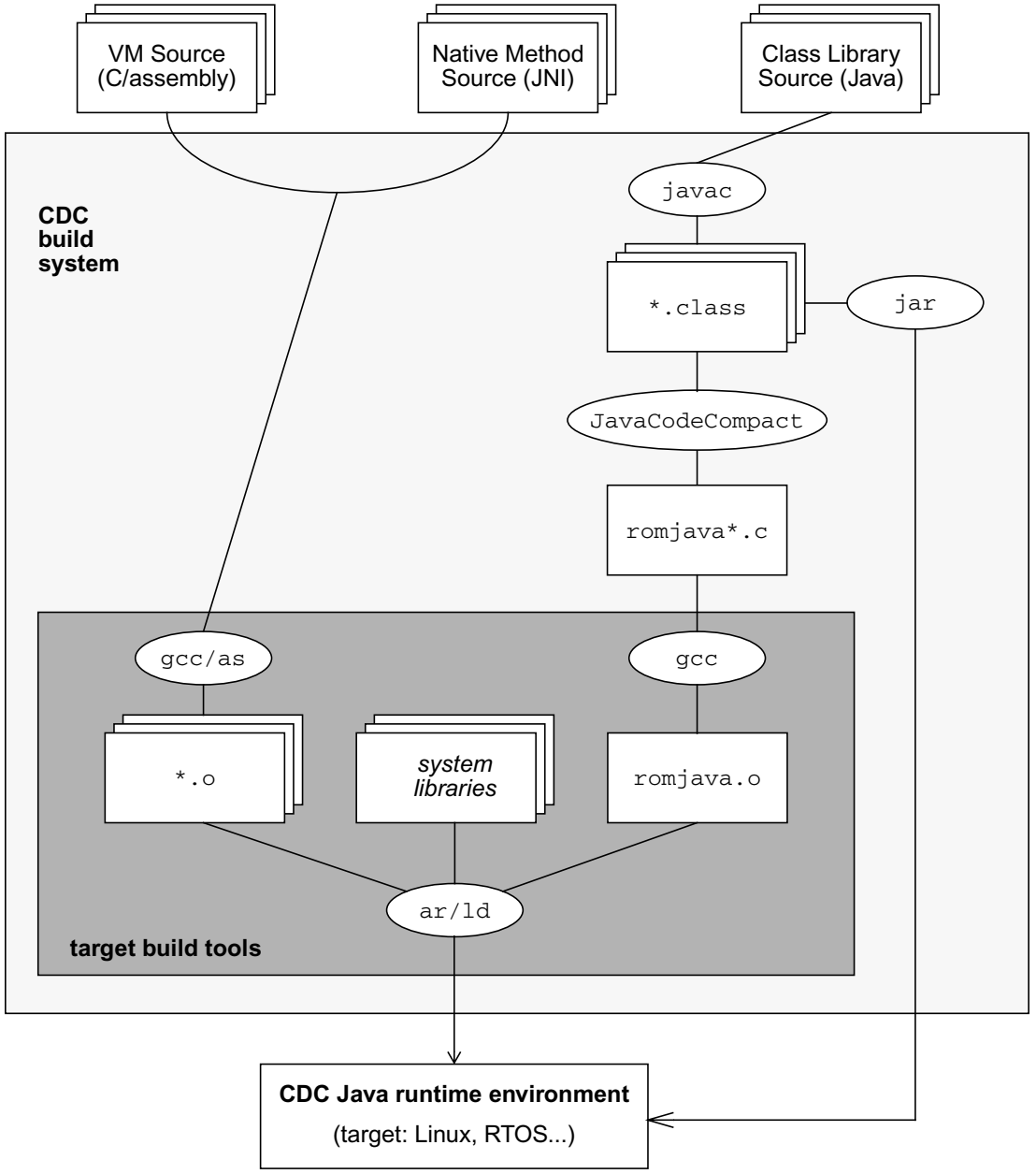

FIGURE 1-1 CDC Build System

CHAPTER 2

Installation

This chapter shows how to install and configure the build system and source code in a CDC source release. The goals here are to:

- Download and install a CDC source release.
- Set up the host and target build tools.
- Configure the CDC build system.
- Test the build system

Chapter 3 describes the contents of the CDC build system. Chapter 4 provides an overview of basic CDC build system procedures. Appendix A provides complete descriptions of the various CDC build options.

2.1 Host Development System Requirements

The CDC build system is based on standard open source and Java build tools, typically hosted on an x86-based PC. In addition, target-specific build tools must be acquired and integrated with the CDC build system. The following subsections describe the system requirements of the CDC build system.

The CDC build system can run on most recent Linux distributions, so this chapter focuses on using the Linux platform as a host development system. Other UNIX platforms can be used without much effort.

TABLE 2-1 describes the basic hardware requirements for the CDC build system on a Linux PC.

TABLE 2-1 Linux PC Host Requirements

Component	Recommended
memory	1-2 GB
disk space	100 GB

In addition to these host requirements, the CDC build system requires a mechanism for downloading a runtime executable to the target device. These mechanisms range from a serial or USB interface to TCP/IP networking.

2.2 Build Tools

The CDC build system is based on commonly available open source and Java build tools. The open source build tools are divided into three categories:

- *Java build tools* are provided by the Java SE SDK and used to build the CDC class libraries as well as some internal build tools like `JavaCodeCompact` and `JavaCodeSelect`.
- *Host build tools* are provided by the host development system for building a few internal build tools and driving the CDC build system.
- *Target build tools* are specific to the target platform.

FIGURE 1-1 describes how these tools are used to build a target-specific version of the CDC Java runtime environment.

The CDC build system was designed to simplify the process of integration with target build tools. If the target build tools are organized and located using common Linux conventions, it should be easy to integrate them with the CDC build tool with the help of a few build options. The easiest way to do this is with the high-level macros `PATH`, `JDK_HOME` and `CVM_TARGET_TOOLS_PREFIX`.

The examples described in the phoneME Advanced Twiki (`http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced`) contain show how to use these basic macros with open source platforms like OpenWRT and commercial platforms like Windows Mobile.

TABLE 2-2 and TABLE 2-4 also describe lower-level macros that define individual tools. These can be used in situations where the high-level macros are insufficient. If the shell locates these tools in `PATH`, only the tool name is needed. Otherwise, it's best to use a full pathname for these macros. Again, the easiest place to define these macros is in the top-level `GNUmakefile`.

2.2.1 Host Build Tools

Host development build tools are used by the CDC build system to build certain host-based CDC build tools. Because these build tools are host-based, they do not create code that is linked into the target runtime system which is usually based on a different target CPU.

Note that in general, the host build tools required by the CDC build system are commonly available on most Linux distributions. In some cases, it may be necessary to download additional packages through a package management system.

TABLE 2-2 describes the macros defined in `build/share/defs.mk` and `build/share/top.mk` that define host build tools.

TABLE 2-2 Host Build Tool Macros

Macro	Default	Description
`HOST_CC`	`$(CC)`	Host C compiler.
`HOST_CCC`	`$(CXX)`	Host C++ compiler.
`LEX`	`lex`	Lexical analyzer.
`BISON`	`bison`	Parser generator.
`MAKE`	`make`	Make utility.
`ZIP` `UNZIP`	`zip`	Zip compression utilities.
`SHELL`	`sh`	Bourne compatible shell[1]

1 See the note in `build/share/defs.mk` about the options for using `ksh` or `sh`.

2.2.2 Java Build Tools

The Java build tools are usually located with the `JDK_HOME` build option. The CDC build system can use the Java SE SDK, version 1.4.2 or later. Sun provides versions of these tools for various development platforms at `http://java.sun.com/j2se/downloads.html`. The CDC build system requires the Java SE version 1.4.2 SDK or later.

TABLE 2-3 describes the standard Java SE build tools and their associated CDC build system macros.

TABLE 2-3 Java SE Build Tools Macros

Macro	Default	Description
JDK_HOME	*unset*	Location of the Java build tools.
CVM_JAVA	if $(JDK_HOME) is set: $(JDK_HOME)/bin/java otherwise search $(PATH) for java	Java application launcher.
CVM_JAVAC	if $(JDK_HOME) is set: $(JDK_HOME)/bin/javac otherwise search $(PATH) for javac	Java compiler.
CVM_JAVAH	if $(JDK_HOME) is set: $(JDK_HOME)/bin/javah otherwise search $(PATH) for javah	JNI C header and stub file generator.
CVM_JAR	if $(JDK_HOME) is set: $(JDK_HOME)/bin/jar otherwise search $(PATH) for jar	Java archive tool.

2.2.3 Target Build Tools

The target build tools include a C/C++ compiler, various binary utilities like an assembler and linker, header files and libraries. The target build tools are usually provided by the CPU/development board vendor or, in cases like OpenWRT, the open source build system generates a custom toolchain for a target device.

Describing how to acquire or build these tools is beyond the scope of this guide, though the phoneME Advanced Twiki (http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced) contains concrete examples of how to setup the CDC build system based on open source and commercial platforms.

TABLE 2-4 describes the top-level and tool-specific macros for the CDC build system.

TABLE 2-4 Target Build Tool Macros

Macro	Default	Description
CVM_TARGET_TOOLS_PREFIX	*unset*	Prefix for the UNIX target build tools.
TARGET_CC TARGET_CCC	$(CVM_TARGET_TOOLS_PREFIX)gcc	The target C/C++ compilers. Note that TARGET_CC and TARGET_CCC are set to HOST_CC and HOST_CCC if CVM_TARGET_TOOLS_PREFIX is not set (or is not valid). The CDC-HI Java virtual machine has been compiled with several versions of the gcc C/C++ compiler.
TARGET_AS	$(TARGET_CC)	The assembler translates assembly language source into a binary format suitable for use by the linker. Note that the assembly language source code provided in the CDC source release is based on the GNU assembler and may need modification to work with a different target assembler.
TARGET_LD	$(TARGET_CC)	The linker combines object and archive files, relocates their data and resolves symbol references.
TARGET_AR	$(CVM_TARGET_TOOLS_PREFIX)ar	The archive utility creates, modifies and extracts object code archives.
TARGET_AR_CREATE	$(TARGET_AR) rc $(1)	Archive creator.
TARGET_AR_UPDATE	$(TARGET_RANLIB) $(1)	Archive updater.
TARGET_RANLIB	$(CVM_TARGET_TOOLS_PREFIX)ranlib	The archive indexer generates an index of an archive's contents and stores it in the archive.

The simplest method is with the CVM_TARGET_TOOLS_PREFIX build option. If the target build tools use a regular naming convention, then the CVM_TARGET_TOOLS_PREFIX build option can locate them for the CDC build system. Note that this macro is not exactly a path. It is a prefix that includes a path and the shared portion of the tool name that precedes the root tool name.

The purpose of CVM_TARGET_TOOLS_PREFIX is to find target build tools with a certain name scheme. For example, the following build option finds tools like arm-linux-gcc and arm-linux-ar and match them to the internal macros used by the CDC build system.

```
CVM_TARGET_TOOLS_PREFIX=/opt/arm-linux/bin/arm-linux-
```

Note the trailing '-'.

If it is not possible to use CVM_TARGET_TOOLS_PREFIX, then the build options described in TABLE 2-4 provide an override mechanism for more precisely specifying the locations of the target build tools. TABLE 2-4 and TABLE 2-2 describe the macros defined in build/share/defs.mk that define the target build tools.

2.3 Build System Setup Examples

The phoneME Advanced Twiki (http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced) contains concrete examples of how to use the CDC build system based on open source platforms like OpenWRT and commercial platforms like Windows Mobile. The Windows Mobile example demonstrates how to use a Windows-hosted build toolchain based on Cygwin (http://www.cygwin.com).

CHAPTER 3

Build System Layout

This chapter describes the contents of the CDC source release from the perspective of the build system. The goals here are to describe the directory structure and makefiles of the CDC build system.

3.1 Relationship to the phoneME Open Source Project

Note that the CDC source code and build system hierarchies have been integrated with the Java ME optional package and profile hierarchies. So the CDC source code and build system are now components within a larger source code repository that is is visible through the phoneME open source project.

The top-level components in the phoneME Project are described in TABLE 3-1.

TABLE 3-1 phoneME Source Repository Components

Component	Description
`cdc`	CDC libraries, VM and build system.
`tools`	Build scripts and tools.
`midp`	MIDP implementation.
`pcsl`	PCSL implementation.
`javacall`	JavaCall porting layer used by MIDP and optional packages.
`jpeg`	JPEG implementation.
`abstractions`	Abstractions shared by optional packages.
`jsr`*`nnn`*	Each component for an optional package implementation has a component name with the form jsrnnn.

In addition, the commercial version of CDC technology has a few source repository components that are not available through the open source project. These are described in TABLE 3-2.

TABLE 3-2 Commercial Source Repository Components

Component	Description
`cdc-com`	Contains a source code overlay for features that are not available through the open source repository.
`secop`	Contains source code and restricted binary implementations for the Security Optional Packages.
`legal`	Copyright and third-party license notices.

3.2 `cdc` Component

The CDC build system is contained in and driven from within the `cdc` component. Inspecting its top-level directories shows how the source code and build system is structured:

- `build` - the CDC build system
- `src` - shared and target-specific source code
- `test` - miscellaneous test programs

This chapter focuses on the `build` directory. For a description of the `src` directory, see the companion document *CDC Porting Guide*. For a description of the various files generated for the CDC Java runtime environment, see the companion document *CDC Runtime Guide*.

3.2.1 `build` Directory Structure

The CDC build system is located in the `build` directory which contains a series of subdirectories that follow the naming conventions described in TABLE 3-3. These subdirectories have parallel organizations to ease navigation and support the operation of the CDC build system.

TABLE 3-3 `build` Directory

Directory	Example	Description
`portlibs`	-	Makefile definitions for the shared JIT layer.
`share`	-	Shared makefiles.

<CPU>	`arm`
	`mips`
	`x86`
		CPU architecture-specific build options. These are mostly JIT-related.

<OS>	`linux`	OS-specific build options for the VM, class library and tools.

<OS>-<CPU>	`linux-x86`
	`linux-arm`
	`linux-mips`
		OS/CPU build options for the VM and tools.

<OS>-<CPU>-<DEVICE>	`linux-x86-suse`
	`linux-arm-zaurus`
	`linux-mips-openwrt`
		The main target build directory. It contains the top-level makefile which can set or override build options used by the shared makefiles. This is also where the generated files are placed. These generated files include the contents of the CDC Java runtime environment and other generated files, depending on which build options are selected.

3.2.2 Makefile Hierarchy

The CDC build system uses the naming convention described in TABLE 3-4 to specify makefile names. The different directories listed above can contain makefiles with identical names. In this case the `share` version will attempt to include the *CPU*, *OS*, *OS-CPU* and *OS-CPU-DEVICE* versions, if they are present. For example, `share/defs.mk` includes `arm/defs.mk`, `linux/defs.mk`, `linux-arm/defs.mk` and `linux-arm-zaurus/defs.mk`.

Profile-based makefiles are chained together. For example, the top-level `share/top.mk` includes `share/defs_`*`classlib`*`.mk` which then includes `share/defs_cdc.mk`.

TABLE 3-4 CDC Build System Makefiles in `build/share`

File	Description
`GNUmakefile`	The top-level makefile for building a runtime environment for a target device.

`defs.mk`
`defs_`*`profile`*`.mk`
`defs_`*`profile_option`*`.mk`
`defs_`*`package`*`_pkg.mk`
	Build option definitions.

`rules.mk`
`rules_`*`profile`*`.mk`
`rules_`*`profile_option`*`.mk`
`rules_`*`package`*`_pkg.mk`
	Makefile rule definitions.

`defs_zoneinfo.mk` `rules_zoneinfo.mk`	`javazic` utility and TimeZone resource files.
`id_`*`profile`*`.mk`	Build identification string definitions.
`jvmti.mk`	Builds JVMTI.
`jcc.mk`	Builds and runs `JavaCodeCompact`.
`jcs.mk`	Builds and runs `JavaCodeSelect`.
`testgc.mk`	GC test framework.
`top.mk`	Top-level shared makefile included by `GNUmakefile` that includes all the other makefiles.

3.3 `secop` Component

The `secop` component contains the source and build system for the Security Optional Packages, which are defined in JSR 219 Foundation Profile.

The `secop` component has two top-level directories:

- `build` - Contains a makefile for building the Security Optional Packages.
- `src` - Contains the source code for the Security Optional Package framework as well as binary plug-ins for the various provider implementations.

The Security Optional Packages are included in a build with the `USE_SECOP` and `SECURITY_PKGS` build options described in Appendix A.

See the *CDC Runtime Guide* for more information about security features.

3.4 Generated Files for the CDC Java Runtime Environment

After a build successfully completes, the target device build directory (e.g. `build/linux-x86-suse`) contains a collection of generated files like object files, executable binaries, Java class files, Zip and `jar` archives. A subset of these files represents a CDC Java runtime environment that can be deployed on a target device while other files contain runtime and application development resources. See Section 4.1.7, "Creating a Runtime Bundle" on page 4-4 for instructions on how to create a bundle containing the CDC Java runtime environment.

The most important runtime files are located in the bin and lib directories. These are described in the companion document *CDC Runtime Guide* which also describes command-line arguments, system properties and other runtime features.

3.4.1 Development Files

The CDC build system generates both a target CDC Java runtime environment and development resources for that target. The table below describes the development resources generated for the target platform.

TABLE 3-5 Generated Development Files

File/Directory	Description
`btclasses.zip` `btclasses/`	`btclasses.zip` contains a version of the CDC class library that can be used for compiling application source code. Since the contents of `btclasses` can vary depending on the selected build options, application development should be based on a target development version of the CDC Java class library. See Section 4.3, "Building a Development Version of the CDC Java Class Library" on page 4-6 for more information.

3.4.2 Test and Demo Programs

A CDC source release includes source code for a collection of test and demo programs that can quickly test the functionality of a CDC Java runtime environment. By default, the CDC build system compiles these test programs and places the compiled class files in the `testclasses` and `democlasses` subdirectories in the

target build directory. For convenience, the build system also creates Zip archives named `testclasses.zip` and `democlasses.zip` that can be easily moved onto a target device for testing.

TABLE 3-6 Test and Demo Files

File/Directory	Description
`democlasses.zip` `democlasses/`	Demo applications that demonstrate profile-based functionality. The source code for these programs is located in `src/share/personal/demo`, `src/share/basis/demo` and `src/share/cdc/demo`.
`testclasses.zip` `testclasses/`	Test applications that can be used to quickly test the CDC Java runtime environment. The source code for these programs is located in `src/share/javavm/test`. The easiest test programs to use are `HelloWorld` and `Test`.

3.4.3 Other Generated Files

In addition, the CDC build system generates other internal object files that are part of the build process.

TABLE 3-7 Other Generated Files

File/Directory	Description
`classes.jcc/`	Compiled class files for `JavaCodeCompact`.
`classes.tools/`	Compiled class for `GenerateCurrencyData` tool.
`generated/`	Miscellaneous generated files.
`jcs/`	`JavaCodeSelect` generated files.
`obj/`	Compiled object files for the VM and class library JNI code.

CHAPTER 4

Build System Procedures

Once installed, the CDC build system can perform a variety of functions. The procedure below describes a simple method for performing a test build to make sure that the CDC build system is correctly installed. Chapter A describes the build options that are available in the CDC build system.

4.1 The Build Cycle

The basic work flow for using the CDC build system is:

Edit source code --> Build with options --> Test

1. **Edit source code.** In this step, you create or modify source code within the target-specific (non-shared) portion of the implementation source code. See the companion document *CDC Porting Guide* for information about how to modify the implementation source code.
2. **Build with options.** In this step, you build a binary executable of the CDC Java runtime environment based on a set of build options specified on the make command line.
3. **Test.** In this step, you launch a Java application using the binary executable running on a target platform.

4.1.1 Performing a Test Build

The target device build directory (e.g. `build/linux-x86-suse`) contains the top-level makefile for building the CDC Java runtime environment for a target device. The example below uses the default values for the build options described in Chapter A.

1. **Change the current directory to the target device build directory:**

   ```
   % cd build/target-platform
   ```

2. **Create a build driver script:**

   ```
   #!/bin/sh
   make \
     JDK_HOME=jdk-dir \
     CVM_TARGET_TOOLS_PREFIX=target-tools-dir
     J2ME_CLASSLIB=foundation
   ```

   Usually, the CDC build options are either included in the `GNUmakefile` or the build driver script.

3. **Build the CDC Java runtime environment:**

   ```
   % sh < build-driver-script.sh
   ```

   When the build is complete, the target device build directory contains the executable binary files for the target platform and other generated files. These generated files are described in Section 3.4, "Generated Files for the CDC Java Runtime Environment" on page 3-5, Section 3.4.1, "Development Files" on page 3-5, Section 3.4.2, "Test and Demo Programs" on page 3-5 and Section 3.4.3, "Other Generated Files" on page 3-6.

   You can override the default values described in Chapter A. For example,

   ```
   % make CVM_DEBUG=true
   ```

   generates the debug version of the build target. Note that `CVM_DEBUG` implicitly selects a number of other build options. Section A.2, "Guidelines for Overriding Build Options" on page A-2 shows how to override build options.

4.1.2 Selecting a Target Device

The CDC build system builds a CDC Java runtime environment for a specific target device. The actual target device is determined by the main target build directory. For example, to build a CDC Java runtime environment for an OpenWRT based router, use `build/linux-mips-openwrt` as the target build directory.

4.1.3 Standard API Choices: Profiles and Optional Packages

The standard API choices available in CDC technology are based on configurations, profiles and optional packages. The *Connected Device Configuration* is chosen by using the CDC build system. The CDC profiles are *Foundation Profile*, *Personal Basis Profile*

or *Personal Profile*. One of these is chosen with the `J2ME_CLASSLIB` build option. Finally, optional packages are integrated into the CDC build system with the `USE_`*optional-package* build options.

Historically, the CDC build system had its own mechanism for integrating optional packages based on the `OPT_PKGS` build option. The integration of CDC technology with optional packages that were previously based on CLDC technology required integration of the CDC build system with the optional package build systems.

Now including an optional package in a CDC build is very simple. For example, the security optional package can be included with a single build option:

```
USE_SECOP=true
```

If the `secop` component is not in the same top-level directory as the `cdc` component a second build option can be used to describe the location of the `secop` component:

```
SECOP_DIR=/home/developer/secop
```

The phoneME Advanced Twiki (`http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced`) has more information about building optional packages, including RMI and JDBC.

4.1.4 Selecting Testing and Performance Features

The CDC build system and source code has a number of testing and performance options. These build options are described in detail in Chapter A.

4.1.5 Quick Rebuilds

The CDC build system maintains some state that can help perform quick rebuilds. To rebuild using the same build flags as the previous build, use the `CVM_REBUILD=true` option. This avoids the need to retype command-line options and avoids the risk of a mistake that results in triggering cleanup actions.

Note – This option does not save the value of any options that specify where tools are located, such as `JDK_HOME` and `CVM_TARGET_TOOLS_PREFIX`.

4.1.6 Generating Verbose Build Logs

By default, the CDC build system prints a build log to the standard error output of the shell. A verbose build log can be generated by setting the `USE_VERBOSE_MAKE` to `true`. For example,

```
% make USE_VERBOSE_MAKE=true >& build.log
```

generates a more verbose build log and redirects it to the file `build.log`.

4.1.7 Creating a Runtime Bundle

After a successful build, the target build directory contains the generated files for a CDC Java runtime environment. The contents of this directory vary according to the build options selected, but for the default case the files described in Section 3.4, "Generated Files for the CDC Java Runtime Environment" on page 3-5 are important. See the companion document *CDC Runtime Guide* for more information about the generated files for the CDC Java runtime environment.

Note – When using CDC build system with the `bin` target to create a runtime bundle, the `JAVAME_LEGAL_DIR` must be set to refer to the top-level `legal` directory.

---

1. **Bundle the CDC Java runtime environment for deployment on the target device.**

   ```
   #!/bin/sh

   make CVM_TARGET_TOOLS_PREFIX=/target-tools-dir/tool-string- \
     JDK_HOME=/usr/java/jdk1.6.0_03 \
     USE_CDC_COM=true \
     J2ME_CLASSLIB=foundation \
     JAVAME_LEGAL_DIR=$CDC_INSTALL_DIR/legal \
     bin
   ```

   The runtime bundle will be generated in the top-level install directory in `../../install`.

2. **Change the current directory to the top-level `install` directory:**

   ```
   % cd ../../install
   ```

   To test the runtime bundle, it must be loaded onto a target platform through some communications mechanism like `ftp`(1). Other techniques for loading the CDC runtime bundle onto the target platform are beyond the scope of this guide.

3. **Copy the runtime bundle onto the test system.**

```
% ftp test-system
...
put cdc-runtime.zip
```

4. **Remotely login onto the test system.**

```
% ssh test-system
```

5. **Unload the runtime bundle.**

```
% unzip cdc-runtime.zip
```

To test the runtime bundle, copy over the `testclasses.zip` and `democlasses.jar` archives and perform the test procedure described in the next section.

4.1.8 Testing the Build

You can test the CDC Java runtime environment by running a sample application with `cvm`, the CDC Java application launcher:

```
% bin/cvm -cp testclasses.zip HelloWorld
Hello world.
% bin/cvm -cp testclasses.zip Test
.............
*CONGRATULATIONS: test Test completed...
```

The source code for these test programs is located in `src/share/javavm/test`.

4.2 JVMTI Support

The CDC HotSpot Implementation now includes support for the new Java Virtual Machine Tools Interface (JVMTI) introduced in JDK 5.0. It provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). JVMTI supports the full breadth of tools that need access to JVM state, including profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

Note – JVMTI replaces the Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI). JVMPI is still available in the CDC HotSpot Implementation as a legacy interface. See Appendix C for a description of JVMPI support.

This chapter describes how to build the CDC Java runtime environment with JVMTI support enabled. See the companion document *CDC Runtime Guide* for information about how to use JVMTI-based tools with the CDC Java runtime environment.

The `CVM_JVMTI` build option enables tool support in the CDC Java runtime environment.

Note – The `CVM_JIT` option must be explicitly disabled for JVMTI builds if it is normally enabled by default.

The steps below demonstrate how to build a CDC Java runtime environment with tool support.

1. **Change the current directory to the target build directory.**

   ```
   % cd build/target-platform-dir
   ```

2. **Build the CDC Java runtime environment with debugging support enabled.**

   ```
   % make CVM_JVMTI=true CVM_JIT=false
   ```

   See the companion document *CDC Runtime Guide* for instructions on how to connect a JVMTI-based developer tool to a CDC Java runtime environment.

4.3 Building a Development Version of the CDC Java Class Library

Note – This section shows how to build a Java class library for a particular CDC implementation. It is meant as a convenience for prototyping. Normal application development should be done with an application developer tool like NetBeans.

When the CDC build system compiles the Java class library for the CDC Java runtime environment, it creates a collection of compiled Java class files that are placed in the `btclasses` and (optionally) *class-lib*`_classes` directories. Because of the way the CDC build system operates, the contents of these compiled class directories can vary based on the selected build options.

Therefore, it is best to create a target development version of the CDC Java class library for each target platform so that it can be used for application development independently of the CDC build system. To do this, use the following build command:

```
% make CVM_PRELOAD_SET=libfull J2ME_CLASSLIB=profile OPT_PKGS=pkgs
```

The `J2ME_CLASSLIB` build option selects a CDC profile and the `OPT_PKGS` build option selects a set of optional packages. The `CVM_PRELOAD_SET=libfull` build option directs the CDC build system to generate a target Java runtime environment with the CDC Java class library entirely in a form that is preloaded and linked with the Java runtime environment. This has the useful side-effect of compiling a version of the CDC Java class library for a target platform that can be easily relocated independently of the Java runtime environment for use with an application development system.

For example, if `J2ME_CLASSLIB=personal` and `OPT_PKGS=rmi`, then the following build command

```
% make CVM_PRELOAD_SET=libfull J2ME_CLASSLIB=personal OPT_PKGS=rmi
```

constructs a file named `btclasses.zip` in the target build directory that contains the compiled CDC Java class library for the target platform containing the Java packages and classes for Personal Profile and the RMI Optional Package.

4.4 Preloading Java Class Files with `JavaCodeCompact`

The CDC build system includes a build tool called `JavaCodeCompact` that reduces the memory needs of a CDC Java runtime environment while improving its performance. `JavaCodeCompact` has its roots in earlier Java technology releases like JavaOS and PersonalJava technologies.

Basically, `JavaCodeCompact` takes platform-independent Java class files and preloads them at build time into a more efficient format that is tightly bound to the VM runtime system. This produces some target-independent C source files whose contents correspond to the virtual machine's runtime data structures that would result if all the classes had been loaded on demand. These source files are then compiled into a platform-specific binary object format and linked with the executable image for the Java runtime environment.

By performing the class loading and linking functions once at build time, `JavaCodeCompact` improves runtime performance and reduces the memory needs of the CDC Java runtime environment. Java classloading semantics are preserved because the runtime system can still load classes and create objects at runtime.

This chapter shows how to use `JavaCodeCompact` within the CDC build system. This includes the following:

- Enabling preloaded builds
- Adding classes to preloaded builds
- Lazy linking support

4.4.1 Linking Java Programs

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

- Use `javac` to compile Java source files into Java class files.
- Load the class files into a Java system, either individually or as part of a `jar` archive.
- Upon demand, the class loading mechanism resolves references to other class definitions.

`JavaCodeCompact` provides an alternate means of program linking and symbol resolution that reduces the VM's resource consumption and improves its performance.

`JavaCodeCompact` performs the following actions during its operation:

- Combines multiple input class files, by combining much of their symbolic information into shared data structures, and concatenating other parts of the classes' definitions.
- Determines the layout and size of all preloaded objects.
- Determines the layout of an object's method table.
- Changes the representation of certain of the Java bytecodes to their "quick" forms.
- Creates header files for use by native code.

4.4.2 Lazy Linking Support

In earlier releases of CDC-HI, the constant pools of preloaded classes could only contain references to other preloaded classes. All the constant pool references had to be fully resolved at build time. This is known as full transitive closure.

CDC-HI now supports unresolved constant pool entries. Preloaded classes can contain references to classes that are later dynamically loaded. This allows a much smaller set of classes to be preloaded

The JCC option `-allowUnresolved` will allow preloaded classes to contain unresolved constant pool entries. The only real behavior change this option causes is to allow unresolved constant pool entries rather than report errors. It does not affect constant pool references to other preloaded classes.

CDC-HI requires that certain classes always be preloaded in order for the for the `cvm` executable to link and run properly. These are classes for which `cvm` has static references. This set of classes is known as the “minimal set”. Use `CVM_PRELOAD_SET=min` to preloaded just this minimal set of classes.

4.4.3 Preloaded Builds

`CVM_PRELOAD_SET` is a build option with several variants that control what will be preloaded. These variants are described in TABLE 4-1.

TABLE 4-1 `CVM_PRELOAD_SET` Build Option Flags

Build Option Flag	Description
`min`	Minimum required classes to build and run the VM.
`minfull`	Same as `min` but with full transitive closure.
`nullapp`	Minimum required classes that avoids any class loading when running an application that does nothing.
`nullappfull`	Same as `nullapp` but with full transitive closure.
`libfull`	All the library classes.
`libtestfull`	All the library and test classes (`testclasses.zip`).

For example, the following `make` command builds a CDC Java runtime environment based on the Foundation Profile with the full Java class library preloaded

```
% make J2ME_CLASSLIB=foundation CVM_PRELOAD_SET=libfull
```

Note that the resulting `bin/cvm` executable is much larger and that the `lib` directory may not contain a `jar` file for the profile. If the `lib` directory does contain a `jar` file, it will include only resource files and not class files. The `bin/cvm` executable contains a preloaded version of the Foundation Profile class library. The size of the preloaded `bin/cvm` is slightly larger than the combination of the non-preloaded `bin/cvm` with the conventionally compiled `lib/`*`class-lib`*`.jar`. But because it can be loaded directly from ROM, the overall memory needs of the Java runtime environment are reduced. Performance is also improved for both launching and operating the CDC Java runtime environment.

4.4.4 Adding Classes to Preloaded Builds

Product-specific classes can be added to the list of preloaded classes. This feature can be used for bundled applications and product-specific class libraries. Because the preloaded classes are linked to the `bin/cvm` executable at build time, this process cannot be undone at a later stage to regain space.

Here's an example of how to add an application class file to the list of preloaded classes:

1. **Compile the Java application.**

   ```
   % javac HelloWorld.java
   ```

2. **Edit** `build/linux-x86-suse/GNUmakefile` **and modify the definition of** `CVM_JCC_INPUT` **to include the compile Java application:**

   ```
   CVM_JCC_INPUT += myclasses/HelloWorld.class
   ```

   `CVM_JCC_INPUT` specifies the list of preloaded classes. The += syntax is necessary to avoid overriding the values defined in `share/jcc.mk`.

3. **Build the CDC Java runtime environment with preloading enabled.**

   ```
   % make
   ```

   By itself, this technique will preload only those classes appended to `CVM_JCC_INPUT`. To preload the full Java class library, use `CVM_PRELOAD_SET=libfull`.

   In this example, it is not necessary to define the class search path for `HelloWorld` with the `-cp` command-line option at runtime because the class has been preloaded. The difference in size between the `cvm` executable is not great because `HelloWorld` is a small class. The benefits of faster launching and operation are more apparent with larger applications.

APPENDIX **A**

Build Option Reference

The CDC build system provides a number of build options that control how a CDC Java runtime environment is built. These include options that are shared across a range of target platforms, like debugging options, profiling options and performance options. At the other end of the spectrum, target-specific options like CPU-specific compiler flags can be specified in the target-specific `GNUmakefile` or in one of the CPU or OS-level makefiles.

A.1 Build Option Categories

This chapter describes the build options found in `build/share`. The most important is `top.mk` which contains the following categories of top-level build options:

- *Fully tested.* Prior to release, the CDC source release undergoes a full QA testing cycle. This testing is based on the default build options, though not all possible combinations have been tested. See the *Release Notes* for a list of fully test build options.
- *Supported.* These build options have been used frequently by the CDC development team, but have not gone through full QA testing.
- *Limited Support.* The default values for these build options are supported. Alternate values have been exercised but should be considered experimental.
- *Deprecated.* These build options have been supported in the past and still may work. But because they are no longer needed because either they have been replaced or are obsolete, they will be removed in a future release.

A.2 Guidelines for Overriding Build Options

Build options can be overridden in several places in the CDC build system. For best results, here are some guidelines for choosing where to override the different kinds of build options.

- Build flags like `CVM_DEBUG` should be overridden on the `make` command-line.
- Target-specific options like `CC_ARCH_FLAGS` and `CC_ARCH_FLAGS_FDLIB` should be set in the `GNUmakefile`.
- Tool configurations can be overridden in `build/`*target*`/defs.mk`.

A.3 Build Option Descriptions

TABLE A-1 and TABLE A-2 describe the top-level build options in the CDC build system.

A.3.1 Supported Build Options

TABLE A-1 Supported Build Options

Build Option	Default	Description
J2ME_CLASSLIB	cdc	The class library build target. The choices are: cdc represents a limited class library that is meant for testing purposes only. foundation represents the full Foundation Profile class library.
SECURITY_PKGS	all	The set of security optional packages. The possible values are: • jaas • jsse • jce • all The syntax of this flag is: SECURITY_PKGS_LIST=<*pkg1*>[,<pkg2,...>]
USE_SECOP	false	Include the Security Optional Packages.
USE_AAPCS	false	*ARM only.* Use the new AAPCS calling conventions instead of the APCS calling conventions.
CVM_DEBUG	false	Build the debug version of the VM. By default, this option enables several other options like CVM_JAVAC_DEBUG.
CVM_DEBUG_ASSERTS	$(CVM_DEBUG)	Enable asserts. Also is forced to true if CVM_VERIFY_HEAP=true.
CVM_DEBUG_CLASSINFO	$(CVM_DEBUG)	Build the VM with the code necessary to interpret class debugging information in the class files. Also causes preloaded classes to include debugging information if they were compiled with it. CVM_JAVAC_DEBUG=true should also be used to provide class debugging information in the CDC and Foundation class files. Otherwise this option will only benefit application classes that are compiled with the -g option.
CVM_DEBUG_DUMPSTACK	$(CVM_DEBUG)	Include support for the CVMdumpStack and CVMdumpFrame functions. CVMdumpStack is useful for dumping a Java stack from gdb after the VM has crashed.

TABLE A-1 Supported Build Options *(Continued)*

Build Option	Default	Description
CVM_DEBUG_STACKTRACES	true	Include code for doing Throwable.printStackTrace and Throwable.fillInStackTrace. If false, then printStackTrace will print a "not supported" message. This is not really just a debug build feature. To slightly reduce the footprint of non-debug builds, set this option to false.
CVM_JAVAC_DEBUG	$(CVM_DEBUG)	Compile classes with debugging information (line numbers, local variables, etc.) by using the -g option. Otherwise build using -g:none. This will not affect the size of the VM image unless CVM_DEBUG_CLASSINFO is also true. Using this option will increase the size of the profile jar file.
CVM_JIT	target-specific: see GNUmakefile	Build a VM with the dynamic compiler.
CVM_AOT	false	Enable runtime support for ahead-of-time compilation (AOTC) of Java methods. See the CDC Runtime Guide for information about how to use the CDC Java runtime environment with AOTC enabled. Requires CVM_PRELOAD_SET=libfull.
CVM_JIT_USE_FP_HARDWARE	target-specific: see GNUmakefile	Enable the dynamic compiler to use an FPU. If true, the dynamic compiler emits FP instructions and uses FP registers. If false, the dynamic compiler stores FP values in general purpose registers and calls out to C or assembler helper functions to do FP arithmetic. NOTE: This option is supported on ARM platforms with vector floating point (VFP) coprocessor support.
CVM_JVMTI	false	Build a VM that supports the new JVMTI debugger/profiler interface. When enabled at runtime, JVMTI will cause a significant degradation of performance.
CVM_JVMTI_ROM	false	Build a VM that supports debugging romized system classes. Note that CVM_JVMTI must also be true.
CVM_OPTIMIZED	!$(CVM_DEBUG)	If true, then use various C compiler optimization features. Setting both CVM_DEBUG=true and CVM_OPTIMIZED=true will provide both debug support and optimized code that will run faster, but not as fast as when using CVM_DEBUG=false.

TABLE A-1 Supported Build Options *(Continued)*

Build Option	Default	Description
CVM_PRELOAD_SET	minfull	Build a VM with the specified set of classes preloaded. Possible choices: • min - minimum required classes to build and run the VM. • minfull - same as min but with full transitive closure • nullapp - minimum required classes that avoids any class loading when running an application that does nothing. • nullappfull - same as nullapp but with full transitive closure • libfull - All the library classes • libtestfull - All the library and test classes (testclasses.zip) Note that full implies full transitive closure of the classes being preloaded, meaning that there will be no references from preloaded classes to classes that need to be dynamically loaded.
CVM_SPLIT_VERIFY	false	Provides faster verification of classes containing StackMap attributes: • Generated by CLDC preverifier tool. • Generated by Java 6 version of javac.
CVM_SYMBOLS	$(CVM_DEBUG)	Include debugging and symbol information for C code even if the build is optimized. Normally, this build option will not affect performance.
CVM_TRACE	$(CVM_DEBUG)	Include support for tracing VM events to stderr. The events that are traced are controlled by the -Xtrace option. Since CVM_TRACE=true slows down the VM substantially, use CVM_TRACE=false and CVM_DEBUG=true to get debugging support without tracing support.
CVM_VERIFY_HEAP	false	Generate verification code for the Java heap. Because this can dramatically affect performance, it can be turned off while still enabling other assertion code with CVM_DEBUG_ASSERTS=true.
CVM_INCLUDE_COMMCONNECTION	false	Include GCF CommProtocol support. This feature is not supported on all platforms.

TABLE A-1 Supported Build Options *(Continued)*

Build Option	Default	Description
CVM_BUILD_SUBDIR_NAME	*unset*	Name of subdirectory to place build into, rather than using the current build directory. Makes it possible to have multiple builds in the same build directory, and makes it easier to remove builds,
CVM_DUAL_STACK	false	Support concurrent CLDC and CDC stacks running on the same VM. This option is most commonly used when supporting a MIDP stack. See the phoneME Advanced Twiki (http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced) for an example of how to build a dual stack enabled version of the CDC Java runtime environment.
USE_VERBOSE_MAKE	false	Avoid printing detailed messages that show each build step. Has the opposite meaning as CVM_TERSEOUTPUT, which can be used instead, but is now deprecated.

A.3.2 Limited Support Build Options

The build options described in TABLE A-2 are limited in that their default values are supported, but alternate values are not and should be considered experimental.

TABLE A-2 Limited Support Build Options

Build Option	Default	Description
CVM_CSTACKANALYSIS	false	Include stub functions to assist in C stack usage analysis.
CVM_GPROF	false	Enable gprof profiling support.
CVM_GPROF_NO_CALLGRAPH	true	When gprof is enabled, this option can be used to control if call graph is generated in the gprof output.
CVM_CCM_COLLECT_STATS	false	Build a VM which collect statistics on the runtime activity of dynamically compiled code, even if the build is optimized.
CVM_CLASSLIB_JCOV	false	Build library classes with -Xjov (JDK 1.4 javac command line option enabled. Also instruments the VM to simulate loading of classfiles for preloaded classes at startup.

TABLE A-2 Limited Support Build Options *(Continued)*

Build Option	Default	Description
CVM_DYNAMIC_LINKING	true	Support the base functionality in the porting layer for dynamic linking. This will be needed by dynamic classloading as well as debugger and profiler support.
CVM_GCCHOICE	generational	Set to the garbage collection technique. semispace, marksweep and generational-seg are unsupported and untested legacy options.
CVM_JIT_DEBUG	false	Build the JIT with extra debugging support, including support for filtering which methods are compiled, and support for tracing the JCS rules used during compilation.
CVM_JIT_ESTIMATE_COMPILATION_SPEED	false	Build a VM which estimates the theoretical maximum compilation speed of the JIT. The measurement is in KB of byte-code compiled per second.
CVM_JIT_PROFILE	false	Enable profiling of compiled code. Use -Xjit:Xprofile=<*filename*> to enable profiling and specify the file to dump profile information. For Linux, enabling profiling at runtime generally degrades performance by about 2%. If profiling support is included at build time but not used at runtime, it has no affect on performance.
CVM_NO_LOSSY_OPCODES	false	Field-related opcodes whose arguments would ordinarily be quickened into offsets instead have their arguments quickened into constant pool references, to ensure the field block for the field is available. This is required to allow the debugger to set field watchpoints. Note this works either with or without classloading enabled, and affects both JavaCodeCompact and quicken.c.
CVM_REBUILD	false	Rebuild using the same build flags as last time, preventing the need to retype a bunch of command line options. The main benefit of this is that there is not risk of having a typo that results in a bunch of cleanup actions triggered. NOTE: this option will not remember the value of any options that specify where tools are located, such as JDK_HOME and CVM_TARGET_TOOLS_PREFIX.

TABLE A-2 Limited Support Build Options *(Continued)*

Build Option	Default	Description
CVM_REFLECT	true	Build a VM that supports the java.lang.reflect package. This does not cause any native function definitions to be eliminated from the build. Instead, their bodies simply throw an UnsupportedOperationException. See the description of CVM_SERIALIZATION for more information. NOTE: setting this option true will result in a VM that is not compliant with the J2ME CDC and Foundation specifications.
CVM_SERIALIZATION	true	Build a VM that supports object serialization (java.io.ObjectInputStream, java.io.ObjectOutputStream). Currently, this only eliminates three functions: JVM_AllocateNewObject, JVM_AllocateNewArray, and JVM_LoadClass0. In addition, serialization depends on reflection, so if CVM_SERIALIZATION is true, CVM_REFLECT will be set to true as well. NOTE: setting this option true will result in a VM that is not compliant with the CDC and Foundation specifications.
CVM_TRACE_JIT	$(CVM_TRACE)	Build a VM with tracing support enabled for all dynamic compiler events, even if the build is optimized. This option is provided to allow building without any other debugging support other than JIT tracing, thus reducing the performance impact. Compiled code will run somewhat slower as a result of the method call tracing that is enabled (estimated 5% slower).
CVM_XRUN	false	Build a VM which supports the -Xrun command-line option for loading native libraries. Defaults to true if CVM_JVMPI is true.
CVM_GCOV	false	Enable gcov code coverage support.

TABLE A-2 Limited Support Build Options *(Continued)*

Build Option	Default	Description
CVM_JIT_PMI	*platform-dependent*	Support for the dynamic patching of calls to methods within compiled methods. Not supported on all platforms. `true` by default for `arm` and `mips` platform when `USE_CDC_COM` is true.
CVM_CREATE_RT_JAR	`false`	By default, non-preloaded JSR and CDC class library are created as separate `jar` files. This option creates a single jar file containing all JSR and CDC class libraries.
CVM_INSPECTOR	`false`	The CVM Inspector is a set of utility functions like `CVMdumpStack()` together with some wrapper functions to make them safe. These can be used to dump information about commonly used VM data structures like thread dumps, heap dumps and object dumps. See the phoneME Advanced Twiki (`http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvancedCVMSH`) for more information about using the CVM Inspector.

TABLE A-3 Legacy Build Options

Build Option	Default	Description
CVM_JVMPI	`false`	Build a VM that supports Java profiling based on JVMPI. This option is not supported with `CVM_JIT=true`. When set `true`, there will be a significant degradation of performance.
CVM_JVMPI_TRACE_INSTRUCTION	`$(CVM_JVMPI)`	Build a VM that supports bytecode tracing for profiling purposes. Enabling this option imposes a greater runtime burden on the interpreter. Hence, this option is provided in case the user does not need this feature and does not want the additional runtime burden to have an impact on the profile they are sampling.

TABLE A-3 Legacy Build Options *(Continued)*

Build Option	Default	Description
CVM_PRELOAD_LIB	*unset*	*Obsolete.* Replaced by CVM_PRELOAD_SET=libfull. Build a VM with all the system and profile classes preloaded. If this build option is set to true then it is equivalent to CVM_PRELOAD_SET=libfull. NOTE: Do not set both CVM_PRELOAD_LIB and CVM_PRELOAD_SET.
CVM_PRELOAD_TEST	*unset*	*Obsolete.* Replaced by CVM_PRELOAD_SET=libtestfull. Build a VM with all the system and profile classes preloaded as well as the test classes (testclasses.zip). If this build option is set to true then it is equivalent to CVM_PRELOAD_SET=libtestfull. NOTE: Do not set both CVM_PRELOAD_TEST and CVM_PRELOAD_SET.
OPT_PKGS		Includes a named optional package in the regular build. The syntax of this flag is: OPT_PKGS=all | <*pkg1*>[,<*pkg2*>] where *pkg1* is the name of the optional package and a ',' is used to separate multiple package names. While this legacy interface can be used for rmi and jdbc, those optional packages are available through USE_RMI and USE_JDBC.

APPENDIX **B**

`JavaCodeCompact` Reference

B.1 Description

`JavaCodeCompact` combines one or more Java class files and produces a target platform-independent C file that contains the given classes in a preloaded format that can be compiled with a C compiler and linked with the executable image of the CDC Java virtual machine. It also provides a way to ensure that certain necessary classes are present and fully linked to expedite the VM's startup and simplify error handling procedures.

B.2 Options

Note – The options described below are for reference purposes only. Setting alternate values for these options is not supported. Only adding class files to `CVM_JCC_INPUT` is supported.

TABLE B-1 `JavaCodeCompact` Options

Options	Description
`filename`	Designates the name of a file to be used as input, the contents of which should be included in the output. File names are not modified by any pathname calculus. File names with a class suffix are read as single class files. File names with `.jar` or `.zip` suffixes are read as Zip files. Class files contained as elements of these files are read and included. Other elements are silently ignored.
`-maxSegmentSize` *num_classes*	Specifies the maximum number of classes to be represented in any one output file. Requires use of the `-o` option to specify output file name. Section B.4, "Output" on page B-4.
`-o` *outfilename*	Provides a template for the name of the output files to be produced. Section B.4, "Output" on page B-4.
`-qlossless`	Preserves more information about the original program in the output file for use of the debugging using the JVMTI debugger interface. See Section B.3, "Opcode Transformations" on page B-3 for a description of the "quickening" process, which is modified by the option. This has a small performance impact on the running system.
`-c`	Performs cumulative linking. Classes that are unresolved by the linking of class files explicitly listed as linker arguments are searched for using the `-classpath` option, and linked as they are found.
`-classpath` *path*	Specifies the path `JavaCodeCompact` uses to look up classes. Directories and Zip files are separated by the delimiter defined by `java.io.File.pathSeparatorChar`, which is generally a colon. Multiple classpath options are cumulative, and are searched left-to-right. This option is only used in conjunction with the `-c` cumulative-linking option.
`-nativesType` *native_type classes*	Indicates the calling convention to be used for native methods of the listed classes. The CNI native type is for use only by classes intimately involved with the virtual machine implementation. All other classes must use the JNI convention. The option sequence "`-nativesType JNI -*`" informs `JavaCodeCompact` of the default type.

TABLE B-1 `JavaCodeCompact` Options *(Continued)*

Options	Description
`-headersDir` *header_type target_directory*	Controls the location of C-language header files generated by `JavaCodeCompact`. Header files for classes of the indicated `header_type` are written in the indicated target directory. Existing header files unchanged remain untouched. A `header_type` is either a `native_type`, as described with the `-nativesType` option above, or an `extra_header_type`, as described below.
`-extraHeaders` *extra_header_typeclasses*	Governs the generation of additional headers for the named classes. The `extra_header_type` of `CVMOffsets` is for use only by classes intimately involved with the virtual machine implementation.
`-v`	Turns up the verbosity of the linking process. This option is cumulative. Currently up to three levels of verbosity are understood. This option is really only of interest as a debugging aid.
`-g`	Enables writing of data information that can facilitate Java debugging, if the information is available in the input data: line-number tables, local variable table and source file names. These tables are not written by default. This option also suppresses the code in-lining optimization.
`-imageAttribute`	Makes all bytecodes writable. By default they are declared as `const`. They must be writable to support breakpointing using the JVMTI debugger interface.
`-noPureCode`	Place all bytecodes in read-write memory. This is useful for setting breakpoints.
`-f` *filename*	Open the named file and read options from it. They are processed just as if they were substituted in the place of this option.
`-allowUnresolved`	Allows deferring of build-time reference checking/linking until runtime. This process is also known as lazy linking. See Section 4.4.2, “Lazy Linking Support” on page 4-8 for more information.
`-cl`	Associates a reuploaded class with a runtime `ClassLoader` instance rather than with just the NULL/bootstrap class loader. This capability can be used with `CVM_DUAL_STACK` to preload a MIDP stack.

B.3 Opcode Transformations

Many Java bytecode instructions refer to symbolic quantities such as the offset of a field or of a method, or to a Java class. Normally, the Java virtual machine resolves such a reference upon first executing the instruction and rewrites the instruction in place. The transformed instruction opcode is referred to as a "quickened" instruction, as subsequent executions of it do not need to see if resolution has taken place, but can proceed assuming it has.

Instead of waiting until runtime to perform this quickening operation, `JavaCodeCompact` "prequickens" each class once at build-time. The result improves classloading performance and makes the resulting code ROMable. A few other transformations take place during linking, including the simple inlining of very short methods.

The usual quickening process makes it harder to reconstruct source code information from the binary program. For example, it is harder to discover name and type information for a class member given only its offset. When retention of this information is important (such as debugging using JVMTI), an alternate set of quickened instructions can be used. They can be more easily interpreted at runtime, but are somewhat slower to execute. This is when `-qlossless` is used.

B.4 Output

The main product of the program is a body of initialized data structures, in C, representing the classes of the input files, and their ancillary data structures, such as Strings, the String intern table, the type table, primitive type classes and many of the array type classes referenced in the input. In addition to one or more `.c` files, a `.h` file is produced called the forward file. This provides forward declarations and is for use only by the other source files produced by `JavaCodeCompact`.

Due to the limitations of many C compilers, it is often necessary to break this output into multiple files. When the `-maxSegmentSize` option is given, multiple `.c` files are produced: one to hold shared data structures such as strings and types, and as many others are necessary, each containing no more than `num_classes` classes.

The names of the files produced are computed using a combination of variables and options.

- *-maxSegmentSize not specified.* If the -o option is given, its argument is used as the name of the single compilable output file. Conventionally, this name ends with .c for C language output, but this is not important to the operation of the program. In the absence of this option, a file is produced with a name based on that of the first input file, stripped of path name prefix and any suffix, to which a `.c` suffix is appended. The resulting name, with "`.h`" appended, is used for the forward file.
- *-maxSegmentSize specified.* The `-o` option must be given in this case. It is used to form this set of file names:
  - `outfilenameList` is an ASCII file naming all the C source programs produced.
  - `outfilenameAux.c` is C file holding data structures not tied to any specific class, such as Strings, String intern table, and the type tables.
  - `outfilenamev.c` For `0 <= v <` (*number of classes*) `/ num_classes`. The C files holding per-class data structures.
  - `outfilename.h` is the forward file name.

CHAPTER C

Legacy JVMPI Support

Note – This appendix contains information about the legacy Java Virtual Machine Profiler Interface (JVMPI), a legacy interface that has been replaced by the Java Virtual Machine Tool Interface (JVMTI). In general, JVMTI should be used instead of JVMPI. The support described here is deprecated and may be removed in a future release.

The CDC HotSpot Implementation supports profiling based on the experimental Java Virtual Machine Profiler Interface (JVMPI) specification. Specifically, the JVMPI-based hprof profiling agent provides reports that include CPU usage, heap allocation statistics and monitor contention profiles.

This chapter describes how to build the CDC Java runtime environment with JVMPI-based profiling enabled. See the companion document *CDC Runtime Guide* for information about how to use the `hprof` Java profiler in a profiling-enabled CDC Java runtime environment.

The `hprof` profiler agent is built into the VM runtime and generates profiling data on the target platform. The source code for the `hprof` is integrated into the VM source directories in `src/share/javavm/{include,runtime}`.

Note – The JVMPI functionality in the CDC HotSpot Implementation is a subset of what the Java SE platform supports. In particular, remote profiling is not supported.

The `CVM_JVMPI` build option enables profiling support in the CDC Java runtime environment. The component that is added to the Java runtime environment that enables debugging support is the JDWP debugging agent.

Note – If `CVM_JIT` and `CVM_JVMPI` are both set to `true`, then the JIT will be disabled at runtime when profiling is used.

The steps below demonstrate how to build a CDC Java runtime environment with Java profiling support.

1. **Change the current directory to the target build directory.**

   ```
   % cd build/linux-x86-suse
   ```

2. **Build the CDC Java runtime environment with debugging support enabled.**

   ```
   % make CVM_JVMPI=true CVM_JIT=false
   ```

3. **Bundle the CDC Java runtime environment for deployment on the target device.**

   ```
   % make bin
   ```

   See the companion document *CDC Runtime Guide* for instructions on how to connect a Java profiler to a Java application running on a CDC Java runtime environment.