CDC Build System Guide

for the Sun Java Connected Device Configuration Application Management System

Version 1.0

Sun Microsystems, Inc.
www.sun.com

November 2005

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 source code organization of the CDC build system and the reference implementations.
- Chapter 4 describes how to use the CDC build system to perform a test build and create a runtime environment deployment bundle, build `javadoc` API reference documentation.
- Chapter 5 describes the configuration options for the CDC build system.
- Chapter 6 describes how to enable debugging support for a CDC Java runtime environment.
- Chapter 7 describes how to enable profiling support for a CDC Java runtime environment.
- Chapter 8 describes how to add an optional package to the CDC build system.
- Chapter 9 describes how to direct the CDC build system to preload the Java class library and application classes.
- Appendix A describes the `JavaCodeCompact` build tool that is used for preloading system and application classes.

Implementation-Specific Features

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

- *Reference Implementation* (RI)
- *Optimized Implementation* (OI)
- *CDC Application Manager* (CAM)

Some topics and discussions in this guide are marked with the acronyms described above to indicate that the material is relevant to a specific CDC technology release.

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

Title	Description
CDC: An Application Framework for Personal Mobile Devices	The white paper *CDC: Java Platform Technology for Connected Devices* introduces CDC technology, standards, devices, applications and tools.
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.

TABLE P-2 Related Documentation

Title	Description
• *CDC Technology Compatibility Kit User's Guide* • *Foundation Profile Technology Compatibility Kit User's Guide* • *Personal Basis Profile Technology Compatibility Kit User's Guide* • *Personal Basis 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 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`.
Inside Java 2 Platform Security	Describes the Java security framework, including security architecture, deployment and customization. See `http://java.sun.com/docs/books/security`.

Accessing Sun Documentation Online

Sun provides online documentation resources for developers and licensees.

TABLE P-3 Sun Documentation Resources

URL	Description
http://docs.sun.com	Sun product documentation
http://java.sun.com/j2me/docs	Java ME technical documentation
http://developer.java.sun.com	Java Developer Services
http://www.sun.com/software/jpe	Java Partner Engineering
http://java.net	An open community that facilitates Java technology collaboration.

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-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. Makefile 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, XScale 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 and TABLE 1-2 describe 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 Build Options

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

- At the top-level, the CDC build system supports several different *target devices*. For example, the CDC build system can generate an executable binary image for one of the target devices described in TABLE 2-6.
- Some build options control *API features* of the generated CDC Java runtime environment. This represents the API target that an application developer refers to when they create application software for the CDC platform.
- Finally, the build options control various *performance and testability options* that are useful during runtime system development. These build options are described in Chapter 5.

1.3 Target Platforms

The implementations of the CDC Java runtime environment described in this guide are based on the target platforms described in TABLE 2-6. These are based on the Linux platform and several embedded processors, including ARM, XScale and MIPS. For help with starter ports that support other operating systems and CPUs, contact Java Partner Engineering (`http://www.sun.com/software/jpe`).

1.4 Selecting 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.

The standard API choices available in CDC 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 4.4, “Standard API Choices” on page 4-3 for information about how to use build options to select standard API featues. See Section 4.12, “Building a Target 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 Chapter 9.

1.6 CDC Build Process Overview

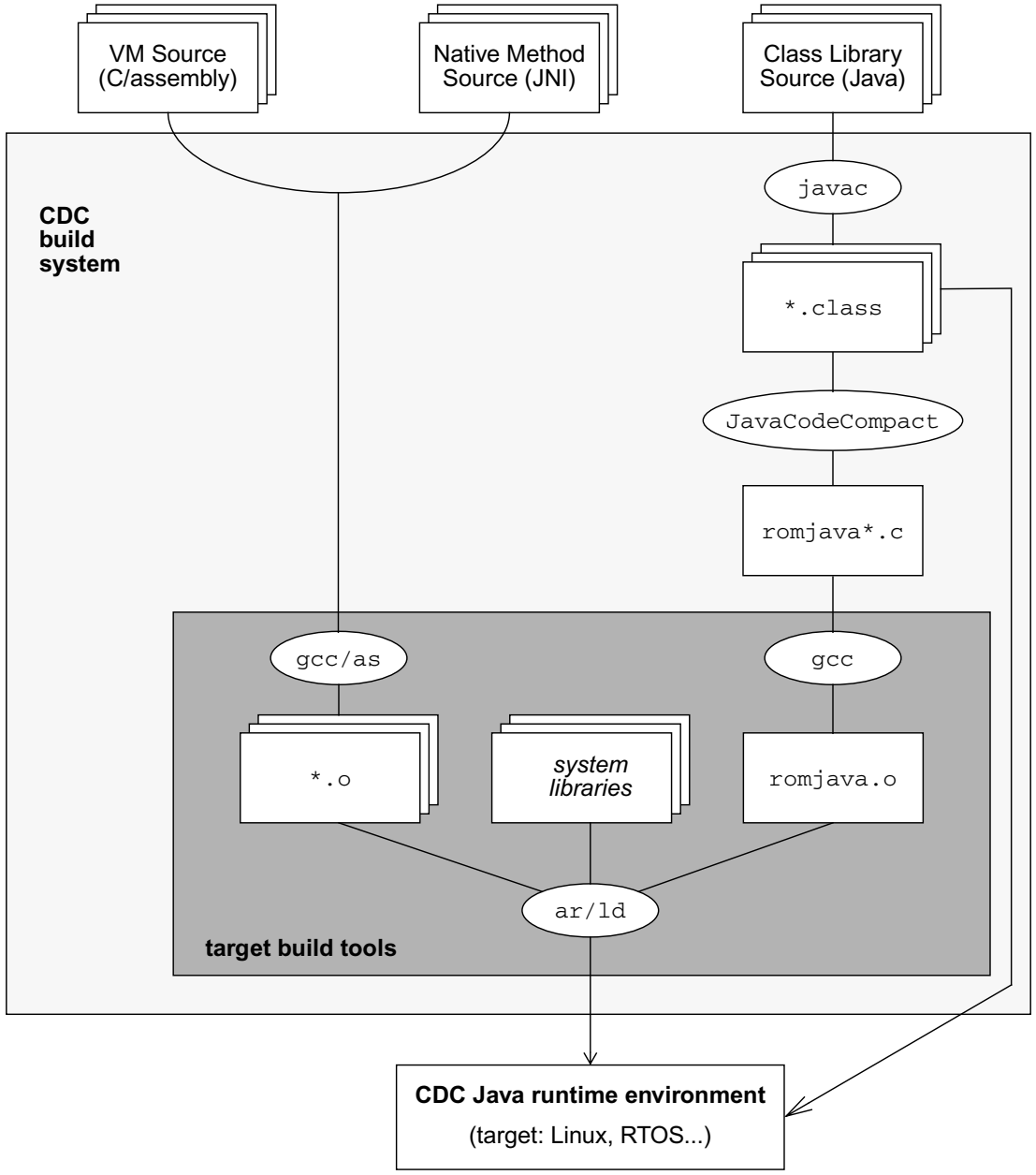

FIGURE 1-1 CDC Build System

The CDC build process is divided into the stages described in TABLE 1-2.

TABLE 1-2 CDC Build Process Overview

Stage	Build Tool	Input	Output
1	Java compiler (`javac`)	The source files for the preloaded system classes are defined in the build macro `CVM_BUILDTIME_CLASSES`.	`build/`*target*`/btclasses/*.class`
2	compression utility (`zip`)	Classes listed in `build/`*target*`/generated/javavm/runtime/tranlist`	`build/`*target*`/btclasses.zip`
3	Java compiler (`javac`)	Java source files for runtime classes	`build/`*target*`/`*class-lib*`/` The *class-lib* directory contains the classes for the CDC Java class library. The name of the directory indicates the supported API(s), e.g. `cdc`, `foundation` or `foundation-rmi`. This directory is not present for preloaded builds.
4	compression utility (`zip`)	Classes contained in `build/`*target*`/`*class-lib*`/`	`build/`*target*`/lib/`*class-lib*`.jar`
5	Java class preloader (`JavaCodeCompact`)	`build/`*target*`/btclasses.zip`	Preloaded class data structures in `build/`*target*`/generated/javavm/runtime/romjava*.c` and `romjava.h`.
6	JNI C header and stub file generator (`javah`)	Runtime classes	JNI header files in `build/`*target*`/generated/jni/*.h`
7	target C compiler	Native methods and VM source (`.c`) files	Object files in `build/`*target*`/obj/*.o`.
8	target C compiler	Preloaded classes in `build/`*target*`/generated/javavm/runtime/romjava*.c`.	Object files in `build/`*target*`/obj/romjava*.o`.
9	target archive utility	Object files in `build/`*target*`/obj/romjava*.o`.	Object archive file in `build/`*target*`/obj/libromjava.a`.
10	target linker	Object files in `build/`*target*`/obj/*.o` from step 8 and the archive file in `build/`*target*`/obj/libromjava.a` from step 10.	The target-specific Java virtual machine executable binary is located in `build/`*target*`/bin/cvm`.

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

After installation, you can familiarize yourself with the contents of the CDC build system by reviewing Chapter 3. Then Chapter 4 provides an overview of basic CDC build system procedures. Then, Chapter 5 provides complete descriptions of the various CDC build options.

2.1 CDC Source Releases

Source code for CDC technology is released in two versions:

- 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, XScale 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.

2.2 CDC Build System Requirements

The CDC build system is based on standard UNIX and Java build tools hosted on an x86-based PC or Solaris workstation. The following subsections describe the system requirements of the CDC build system.

2.2.1 Hardware Requirements

The CDC build system supports two host environments: a Solaris workstation or an x86-based PC. The basic hardware requirements for the CDC build system are described in TABLE 2-1 and TABLE 2-2:

TABLE 2-1 Solaris Host Requirements

Component	Requirement
Solaris operating environment	Solaris 8 or Solaris 9
memory	• minimum: 256MB • recommended: 512MB for a full debugging environment
disk space	• minimum: 100MB • recommended: 5+GB

The x86-based PC host requirements are mostly defined by the need to run Suse Linux or Windows. See the Suse Linux desktop system requirements site (`http://www.suse.com/us/business/products/sld/system_requirements.html`) for a description of the hardware requirements for Suse Linux.

TABLE 2-2 Linux Host Requirements

Component	Description
memory	• minimum: 64MB • recommended: 512MB for a full debugging environment
disk space	• minimum: 100MB • recommended: 5+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. The examples in this guide are based on `ftp`(1).

2.2.2 Software Requirements

The CDC build system is based on commonly available UNIX and Java build tools. The UNIX build tools are divided into two categories: *target build tools* that must be reconfigured and rebuilt for each target platform and *host build tools* that should work without modification on the host development system. In addition, the CDC build system includes some internal build tools like `JavaCodeCompact` and `JavaCodeSelect` as well as standard Java SE build tools like `javac`.

2.2.2.1 UNIX Build Tools

TABLE 2-3 describes the UNIX target build tools. See TABLE 2-7 for a description of the build tool versions for specific platforms.

TABLE 2-3 UNIX Target Build Tools

Build Tool	Example	Description
C/C++ cross-compiler	`gcc`	The reference source code for the CDC-based Java runtime environment and build system has been compiled with several versions of the `gcc` C/C++ compiler.
Assembler	`as`	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.
Linker	`ld`	The linker combines object and archive files, relocates their data and resolves symbol references.
Archive utility	`ar`	The archive utility creates, modifies and extracts archives.
Archive indexer	`ranlib`	The archive indexer generates an index to the contents of an archive and stores it in the archive.

TABLE 2-4 describes the UNIX host build tools for the x86/Suse Linux host development platform.

TABLE 2-4 UNIX Host Build Tools

Tool	Version	Description
`gcc/g++`	3.3.3	host GNU C/C++ compiler[1]

`ar`
`as`
`ld`
`ranlib`
	2.15.90	host GNU binary utilities[1]

`make`	3.80	GNU `make` utility
`sh`	2.05	Bourne compatible shell (e.g. `bash` or `ksh`)
`lex`	2.5.4	lexical analyzer generator
`bison`	1.875	parser generator
`zip`	2.3	Zip compression utility

1 UNIX host development build tools (GNU C/C++ and GNU binary utilities) 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. Most recent versions of the GNU C/C++ compiler after 2.95 should work with the CDC build system.

2.2.2.2 Java SE Build Tools

TABLE 2-5 describes the standard Java SE build tools.

TABLE 2-5 Java SE Build Tools

Tool	Description
`java`	Java application launcher
`javac`	Java compiler
`javadoc`	Java API documentation generator
`jar`	Java archive tool
`javah`	JNI C header and stub file generator

2.2.2.3 Qt Library

The CDC build system can build both the Personal Basis Profile and Personal Profile build targets. These build targets can be based on several different GUI toolkits. The CDC reference implementation requires Qt/X11 version 3.3.1 which is is included in the Suse Linux 9.1 desktop distribution.[1]

2.3 CDC Target Platform Requirements

The CDC Java runtime environment can support a variety of target devices. This section describes the target devices that have been tested with the RI and OI.

TABLE 2-6 Target Platforms

Target Platform	URL	Description
ARM-based Linux	http://www.myzaurus.com	The Sharp Zaurus SL-C860 is a Linux/ARM-based PDA.
	http://www.intel.com/design/pca/applicationsprocessors/tools_software/devboards.htm	The Intel® PXA27x Processor Developer's Kit (Mainstone III) includes a Linux-based Intel XScale PXA270 Processor Card.
MIPS-based Linux	http://www.linux-mips.org/wiki/index.php/Cobalt	The Cobalt Qube 2 is a Linux/MIPS-based server.

For information on the system requirements for supporting alternate target devices, see the *CDC Porting Guide*.

2.3.1 ARM Floating Point Note

The CDC HotSpot Implementation Java virtual machine can support floating point (FP) operations with both a target device's floating point hardware and with software emulation. Most ARM processors (such as StrongARM) do not have FP hardware support. When an FP instruction is encountered in the instruction stream on Linux/ARM, an illegal instruction trap occurs and the Linux kernel emulates the FP instruction. This results in highly inefficient code for FP operations.

To avoid this, the CDC HotSpot Implementation Java virtual machine uses a software floating point model to achieve better performance with software FP emulation. Instead of generating ARM FP instructions, the VM generates runtime calls for FP operations. Floating point parameters are passed in integer registers. This scheme avoids kernel traps for each FP operation.

1. The latest version of the Qt library tested with the CDC source release is version 3.3.4 which is more recent than the version included with the Suse Linux 9.1 desktop distribution. The source bundle for Qt 3.3.4 can be aqcquired from ftp.trolltech.com/pub/qt/source. Licensees may only use this Sun Microsystems, Inc. software with commercial versions of TrollTech's Qt libraries, either provided herewith by Sun or licensed directly by Licensee from TrollTech. Additional Qt header files must be licensed from TrollTech under a TrollTech developer's license.

The target C compiler should be able to perform software floating point compilations. With the `gcc` 2.95.3 compiler, for example, this is achieved by using the `-msoft-float` option.

Note – The CDC build system is based on a patched `gcc` 2.95.x compiler. For performance reasons, the CDC build system uses the `-msoft-float` option for the `linux-arm` builds. The CDC build system can use the `gcc` 3.x compiler for platforms that have been updated to include softfloat libraries. If the target platform has hard float libraries, then `gcc` 2.95.x must be used. Otherwise, `gcc` 3.x can be used. For guidelines on using alternate compilers and build tools, see the *CDC Porting Guide*.

2.3.2 Cobalt Build Notes

The libraries and header files for the Cobalt Qube are older than the current versions of the UNIX build tools support. Therefore, it is necessary to use the libraries and header files from the Cobalt Qube target device. See Section 2.4.4, "Acquiring the UNIX Build Tools" on page 2-7 for instructions on how to configure the UNIX build tools to support a target device with cross-compilation.

2.4 Installation Procedure

The procedures below show how to acquire, install, configure and test the CDC build system for a specific target device.

2.4.1 Downloading the CDC Distribution File

The CDC source code release is available under license from Sun Microsystems from the Java Partner Engineering website at `http://javapartner.sun.com`. For an account name and password for this Web site, contact Java Partner Engineering (`http://wwws.sun.com/software/jpe`).

2.4.2 Extracting the Distribution Bundle

The CDC distribution bundles are delivered in Zip format and can be unloaded with the `unzip`(1) command.

1. **Change the current directory to a location in a file system with enough free disk space to hold the CDC source release.**

   ```
   % cd /net/cdc-build
   ```

2. **Download the distribution bundle from the Java Partner Engineering download site.**

3. **Unzip the distribution bundle:**

   ```
   % unzip cdc-1_1-src-linux-x86.zip
   ```

   After unloading the distribution file, you can browse through the source and build hierarchies. For a description of the main directories of the CDC build system, see Chapter 3. For a description of the different runtime files generated by the CDC build system, see the *CDC Runtime Guide*. For a description of the source code files and directories, see the *CDC Porting Guide*.

2.4.3 Acquiring the Java Build Tools

The CDC build system uses the Java build tools described in TABLE 2-5. 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.

2.4.4 Acquiring the UNIX Build Tools

The host UNIX build tools supplied by the host development environment can be used without modification. It will be necessary to acquire and build the target build tools separately for each target platform.

TABLE 2-7 describes the build tool versions used with the CDC build system to test and validate the CDC Java runtime environment on the supported platforms.

TABLE 2-7 Supported Target Build Tool Versions

Platform	Compiler (`gcc`)	Binary Utilities (`ar`, `as`, `ld`, `ranlib`)
ARM/Zaurus	2.95.3 (patched)	2.11
ARM/Bulverde	3.2.3 (patched)	2.14.90.0.7 (patched)
MIPS/Cobalt	3.4.2	2.15

For both Solaris and Linux, the procedure of acquiring the target build tools is the same:

1. **Copy the libraries and header files from the target platform for use with the target compiler on the host development system.**

2. **Configure, build and install the target build tools.**

   See the *GNU Compiler Collection* documentation (`http://gcc.gnu.org/onlinedocs`) for instructions on how to configure, build and install the cross-development build tools for a given target platform. For example, the `configure` build script has a `--with_cpu` command-line argument. Passing the `strongarm` option to this command-line argument adds support for the StongARM CPU. After the target build tools have been built, they should be installed in a location for use with the rest of the CDC build system.

2.4.4.1 `gcc` 2.95.3 Notes

- Processors like StrongARM may not have hardware-based floating point support. Build and install the SoftFloat library from `http://www.jhauser.us/arithmetic/SoftFloat.html`.
- Apply the `gcc` patch from `http://handhelds.org/download/toolchain/source/gcc-2.95.2-diff-991022.gz`. This patch improves support for certain processors like ARM.

2.4.4.2 XScale/Bulverde Notes

The target build tools for the XScale-based Mainstone III development board are described in TABLE 2-8. These build tools require some patches included in the patch files described in TABLE 2-8 as well as some additional patches described below.

TABLE 2-8 XScale/Bulverde Target Build Tools

Build Tool	URL
Binaries	ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/xscale/mainstone/12-29-2004/bin/arm-linux-toolchain-bin-11-26-04.tar.gz
Patches	ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/xscale/mainstone/12-29-2004/src/toolchain/patch/binutils.patch
	ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/xscale/mainstone/12-29-2004/src/toolchain/patch/gcc-base.patch
Source	ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/xscale/mainstone/12-29-2004/src/toolchain/source/binutils-2.14.90.0.7.tar.gz
	ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/xscale/mainstone/12-29-2004/src/toolchain/source/gcc-3.3.2.tar.gz

1. **Download the target build tool sources and apply the patches described in TABLE 2-8.**

2. **Apply the patch below to `gcc/config/arm/t-linux` to disable the ASM soft float library functions:**

```
-LIB1ASMFUNCS = _udivsi3 _divsi3 _umodsi3 _modsi3 _dvmd_lnx \
-    _negdf2 _addsubdf3 _muldivdf3 _cmpdf2 _unorddf2 _fixdfsi
_fixunsdfsi \
-    _truncdfsf2 _negsf2 _addsubsf3 _muldivsf3 _cmpsf2 _unordsf2 \
-    _fixsfsi _fixunssfsi
+LIB1ASMFUNCS = _udivsi3 _divsi3 _umodsi3 _modsi3 _dvmd_lnx
```

3. **Apply the patch below to `gcc/config/arm/t-linux` to enable C soft float library functions:**

```
+FPBIT = fp-bit.c
+DPBIT = dp-bit.c
+
+fp-bit.c:$(srcdir)/config/fp-bit.c
+    echo '#define FLOAT' > fp-bit.c
+    cat $(srcdir)/config/fp-bit.c >> fp-bit.c
```

```
+dp-bit.c:$(srcdir)/config/fp-bit.c
+    cat $(srcdir)/config/fp-bit.c >> dp-bit.c
+TARGET_LIBGCC2_CFLAGS = -Dinhibit_libc -fno-inline
```

Note that the gcc patching and rebuilding is only necessary to make sure all TCK Floating Point tests pass. If the user just wants to experiment with the platform, the existing Mainstone gcc binaries should work fine and probably will not result in any incorrect behaviour in the typicial java application.

2.4.5 Organizing the CDC Build Tools

The CDC build system uses a set of macros that define the locations of the UNIX and Java build tools. These macros allow the CDC build tools to be organized in different ways.

The following strategies can be used for organizing the CDC build tools:

- Use the default locations.
- Redefine the top-level macros to place the build tools in alternate locations.
- Override the individual build tool macros listed in TABLE 2-11 and TABLE 2-12.

These strategies provide flexibility in organizing the CDC build tools without relying on the `PATH` environment variable.

2.4.5.1 Using the Default Locations

The easiest strategy is to use the default locations:

1. **Use `/micro/tools` as the root of the build tool hierarchy.**
2. **Install the UNIX cross-development build tools in `/micro/tools/`*<host_cpu>*`-`*<host_device>*`-`*<host_os>*`/gnu/bin/`*<target_cpu>*`-`*<target_device>*`-`*<target_os>*`-`*<tool>*.**

   For example, the `gcc` compiler would be located in `/micro/tools/sparc-sun-solaris/gnu/bin/x86-suse-linux-gcc`.
3. **Install the Java build tools in `/micro/tools/`*<host_cpu>*`-`*<host_device>*`-`*<host_os>*`/java/jdk1.4.2`.**

   For example, the Java build tools would be located in `/micro/tools/sparc-sun-solaris/java/jdk1.4.2`.

2.4.5.2 Redefining the Top-Level Macros

TABLE 2-9 describes the top-level macros used to organize the CDC build tools. Overriding these macros is not necessary if the build tools are installed in the default location where the CDC build system can easily find them.

The macros described below and in Chapter 5 can be overridden on the `make` command-line, in the `GNUmakefile` in the target device build directory, or in a `defs.mk` file in the target device build directory..

TABLE 2-9 Cross-Development Tool Macros

Macro	Default	Description
`CVM_TOOLS_DIR`	`/micro/tools`	The top-level directory for the CDC build system tools.
`JDK_VERSION`	`jdk1.4.2`	Version of the Java SE build tools.
`JDK_HOME`	`$(CVM_TOOLS_DIR)/$(CVM_HOST)/java/$(JDK_VERSION)`	Location of the Java build tools. See TABLE 2-10 for a description of `CVM_HOST`.
`CVM_TARGET_TOOLS_DIR`	`$(CVM_TOOLS_DIR)/$(CVM_HOST)/gnu/bin`	Location of the UNIX target build tools.
`CVM_TARGET_TOOLS_PREFIX`	`$(CVM_TARGET_TOOLS_DIR)/$(TARGET_CPU_FAMILY)-$(TARGET_DEVICE)-$(TARGET_OS)-`	Prefix for the UNIX target build tools.

TABLE 2-10 describes some of the internal build system macros. These macros cannot be overridden because they are discovered by the build system through the build directory naming conventions. They are used by the macros in TABLE 2-9 and are listed here for reference purposes.

TABLE 2-10 Internal Build System Macros

Macro	Sample Values	Description
`HOST_CPU_FAMILY`	`x86` or `sparc`	Host CPU.[2]
`HOST_DEVICE`	`suse` or `sun`	Host device.[1]
`HOST_OS`	`linux` or `solaris`	Host operating system.[1]

TABLE 2-10 Internal Build System Macros

Macro	Sample Values	Description
TARGET_CPU_FAMILY	arm[1] mips x86	Target CPU.[3]
TARGET_DEVICE	zaurus xscale suse	Target device.[2]
TARGET_OS	linux	Target operating system.[2]

1 Supports both the ARM and XScale embedded CPU.

2 Used to form CVM_HOST which describes the host platform.

3 Used to form CVM_TARGET which describes the target platform.

The macros described in TABLE 2-9 and TABLE 2-10 are defined in build/share/defs.mk.

Examples for overriding these macros include:

- An alternate location for the Java build tools can be specified with the JDK_HOME macro.
- An alternate root for the build tool hierarchy can be specified with the CVM_TOOLS_DIR macro.
- An alternate location for the UNIX cross-development build tools can be specified with the CVM_TARGET_TOOLS_DIR macro.

2.4.5.3 Overriding the Individual Build Tool Macros

TABLE 2-11 and TABLE 2-12 describe the macros defined in build/share/defs.mk that define individual target build tools. When these macros are overridden, the top-level macros like CVM_TOOLS_DIR and JDK_HOME are ignored.

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.

TABLE 2-11 Target Build Tool Macros

Macro	Default	Description
CC	$(CVM_TARGET_TOOLS_PREFIX)gcc	C compiler
CCC	$(CC)	C++ compiler

TABLE 2-11 Target Build Tool Macros

Macro	Default	Description
AS	$(CC)	assembler
LD	$(CC)	linker
CVM_USE_NATIVE_TOOLS	The default value is based on whether the native tools are found. This is usually true for the RI and false for the OI.	When true, the native tools in $PATH are used rather than attempting to locate a gcc compiler. This means cc is used as the default compiler. When false, the makefiles search for gcc in a path determined by a number of other options, including CVM_TOOLS_DIR and CVM_HOST. See CVM_TARGET_TOOLS_PREFIX in build/share/defs.mk.

TABLE 2-12 describes the macros defined in build/share/defs.mk that define host build tools.

TABLE 2-12 Host Build Tool Macros

Macro	Default	Description
CVM_JAVA	java	Java application launcher
CVM_JAVAC	javac	Java compiler
CVM_JAVADOC	javadoc	Java API reference generator
CVM_JAVAH	javah	JNI C header and stub file generator
CVM_JAR	jar	Java archive tool
AR	ar	archive utility
RANLIB	ranlib	archive indexer
HOST_CC	$(CVM_HOST_TOOLS_PREFIX)gcc or $(CC) when $(CVM_USE_NATIVE_TOOLS) is true.	host C compiler
HOST_CCC	$(HOST_CC)	host C++ compiler
LEX	lex	lexical analyzer
BISON	bison	parser generator
ZIP	zip	Zip compression utility
SHELL	sh	Bourne compatible shell[1]

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

CHAPTER 3

Build System Contents

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.

After the CDC source release bundle has been unzipped and installed, the source code is structured into a few high-level directories:

- `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*.

3.1 `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 3-1. These subdirectories have parallel organizations to ease navigation and support the operation of the CDC build system.

TABLE 3-1 `build` Directory

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

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

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

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

<OS>-<CPU>-<DEVICE>	`linux-x86-suse`
	`linux-arm-zaurus`
	`linux-arm-xscale`
	`linux-mips-cobalt`
		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 Makefile Hierarchy

The CDC build system uses the naming convention described in TABLE 3-2 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 3-2 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`
	Macro 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.

`jdwp.mk`
`jdwp_transport.mk`
`jdwp_transport_socket.mk`
	Java debugger wire protocol makefiles. See Chapter 6 for a description of the debugging support in the CDC build system.

`hprof.mk`	Builds and runs `hprof`, the Java profiler agent.
`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 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 4.10, "Creating a Runtime Bundle" on page 4-5 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 Generated 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 3-3 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 4.12, "Building a Target Development Version of the CDC Java Class Library" on page 4-6 for more information.
`hprof/`	Compiled class files, header files and object files for the HPROF module. See Chapter 7 for a description of the profiling support in the CDC build system.
`jdwp/`	Compiled class files, header files, object files and libraries for the JDWP module. See Chapter 6 for a description of the debugging support in the CDC build system.

3.5 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 3-4 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`.

3.6 CDC AMS Generated Files

The CDC build system generates several files and directories for CDC AMS (`J2ME_PLATFORM=appmanager`).

TABLE 3-5 Other Generated Files

File/Directory	Description
`appmanager.par`	A PAR file containing sample applications for OTA deployment. The `AwtPDA_APPS` macro in `build/share/defs_appmanager_AwtPDA_pmode.mk` can be used to add applications for OTA deployment.
`appmanager_classes/`	Class files built for CDC AMS, including both the presentation mode and the core CDC AMS implementation.
`jaxp_classes/`	Class files built for the XML parser.
`oma-adapter_classes/`	OMA adapter classes included in the EAR file for the J2EE Client Provisioning server.
`par/`	Staging directory for the PAR file target.
`war/`	Staging directory for the WAR file target.

3.7 Other Generated Files

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

TABLE 3-6 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 5 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.

4.2 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 makefile options described in Chapter 5.

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

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

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

```
% make
```

Note – During the build process, the CDC build system displays warning messages about deprecated methods and the missing `empty.mk` file. These warning messages are produced by all supported tool sets and have been investigated and found to be benign.

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 3.3, "Generated Files for the CDC Java Runtime Environment" on page 3-3, Section 3.4, "Generated Development Files" on page 3-4, Section 3.5, "Test and Demo Programs" on page 3-4 and Section 3.7, "Other Generated Files" on page 3-6.

You can override the default values described in Chapter 5. For example,

```
% make CVM_DEBUG=true
```

generates the debug version of the build target. Note that `CVM_DEBUG` implicitly selects a number of other makefile options. Section 5.2, "Guidelines for Overriding Makefile Options" on page 5-2 shows how to override makefile options.

4.3 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 a Suse Linux-based x86/PC, use `build/linux-x86-suse` as the target build directory.

4.4 Standard API Choices

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. One of the CDC profiles like *Foundation Profile*, *Personal Basis Profile* or *Personal Profile* is chosen by using one of the different source releases based on a CDC profile and using `J2ME_CLASSLIB`. And optional packages like RMI and JDBC are selected by integrating their source bundles into the CDC build system and using the `OPT_PKGS` and `SECURITY_PKGS` build options. See Chapter 8 for information about how to add an optional package to the CDC build system.

4.5 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 5.

4.6 Building CDC AMS

CDC AMS has a few special build targets and build options. The three most important build options for building CDC AMS are:

- `J2ME_PLATFORM=appmanager`
- `PRESENTATION_MODES` (`PBP` for Cobalt, `AwtPDA` for Zaurus)
- `QTEMBEDDED` (optional)

The first two are necessary for any CDC AMS build. `J2ME_PLATFORM` is the main CDC AMS build option. `PRESENTATION_MODES` specfies the presentation mode build target. When choosing a value for `PRESENTATION_MODES`, it is necessary to choose an appropriate profile target like `J2ME_CLASSLIB=personal`.

Note – `QTEMBEDDED` is not specifically a CDC AMS build option, but it is necessary to build for the Zaurus target.

4.7 Building OTA Support for CDC AMS

CDC AMS includes support for OTA provisioning based on the J2EE Client Provisioning Server RI (CPRI). To enable use of this feature in the CDC Java runtime environment, both the CPRI and the J2EE 1.3.1 SDK distribution bundles must be integrated with the CDC build system. This integration is necessary so that the CDC AMS client includes an OTA adapter and so that the Java build system can create an EAR file for deployment in a J2EE server.

The following procedure shows how to acquire the necessary source bundles for CPRI and J2EE 1.3.1 SDK and build an EAR file for deployment in a J2EE server.

1. **Download and install the J2EE 1.3.1 SDK from** `java.sun.com/j2ee/1.3/download.html`**.**
2. **Download and unbundle the CPRI source bundle from** `java.sun.com/j2ee/provisioning/download.html`**.**
3. **Build a client version of CDC AMS as shown in** **Section 4.6, "Building CDC AMS" on page 4-3****.**
4. **Build an EAR file by using the previous step and adding the** `web` **build target and the** `J2EE_HOME` **and** `JSR124_HOME` **macros.**

   ```
   % make web J2EE_HOME=j2ee_home JSR124_HOME=jsr124_home
   ```

   The web.ear file can be deployed in the J2EE server with a procedure described in the *CDC Java Runtime Guide*.

4.8 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_TOOLS_DIR`.

4.9 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 `CVM_TERSEOUTPUT` to `false`. For example,

```
% make CVM_TERSEOUTPUT=false >& build.log
```

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

4.10 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 makefile options selected, but for the default case the files described in Section 3.3, "Generated Files for the CDC Java Runtime Environment" on page 3-3 are important. See the companion document *CDC Runtime Guide* for more information about the generated files for the CDC Java runtime environment.

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

   ```
   % make bin
   ```

   The runtime bundle is kept 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 rt.tar.gz
   ```

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

   ```
   % ssh test-system
   ```

5. **Unload the runtime bundle.**

   ```
   % tar xvzf rt.tar.gz
   ```

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

4.11 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`.

4.12 Building a Target Development Version of the CDC Java Class Library

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_LIB=true 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_LIB` 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_LIB=true 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.

4.13 Building `javadoc` API Reference Documentation

The CDC build system can generate `javadoc` API reference documentation. This can be useful in cases where a licensee either adds functionality to a Java runtime environment, perhaps with a product-specific package (e.g. `com.myproduct.*`) or to generate a `javadoc` bundle that more accurately reflects the set of optional packages in a CDC-based product.

The command for generating `javadoc` reference documentation is:

```
% make J2ME_CLASSLIB=profile javadoc.zip
```

The `javadoc`-generated HTML files are place in the `install` directory.

Note – The `javadoc` source for the Personal Basis Profile class library and the Personal Profile class library is not complete or accurate. Attempting to generate javadoc reference documentation for these build targets will generate an error message from the CDC build system. Refer to the JCP specifications for Personal Basis Profile (JSR-217) and Personal Profile (JSR-216) for API reference documentation.

CHAPTER 5

Makefile Options and Macros

The CDC build system provides a number of makefile options and macros 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.

5.1 Makefile Option Categories

This chapter describes the makefile options found in `build/share`. The most important is `top.mk` which contains three categories of top-level makefile 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 makefile 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 options are supported. Alternate values have been exercised but should be considered experimental.

5.2 Guidelines for Overriding Makefile Options

Makefile 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 makefile 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`.

5.3 Makefile Option Descriptions

TABLE 5-1 and TABLE 5-2 describe the top-level makefile options in the CDC build system.

5.3.1 Supported Makefile options

TABLE 5-1 Supported Makefile Options

Makefile Option	Default	Description
J2ME_CLASSLIB	cdc	The class library build target. The possible values are: • cdc • foundation • basis • personal
AWT_IMPLEMENTATION	qt (PBP) peer_based (PP)	Specifies which implementation of AWT to build.
AWT_PEERSET	qt (PP)	Specifies which AWT native bridge to build.
QTEMBEDDED	false	Builds with the Qt Embedded library. Note that this build option is overridden with a value of true for Zaurus builds.
SECURITY_PKGS		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*,...>]
J2ME_PLATFORM		[CDC *AMS only.*] appmanager
PRESENTATION_MODES		[*CDC AMS only.*] AwtPDA, PBP, all
CVM_MTASK	true	[CDC *AMS only.*] Include the mtask process-based application management system.
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. When OPT_PKGS is set to all, all available optional packages will be part of the compilation. See Chapter 8 for more information on including optional packages in a build.
CVM_DEBUG	false	Build the debug version of the VM. By default, this option enables several other options like CVM_JAVAC_DEBUG.

TABLE 5-1 Supported Makefile Options

Makefile Option	Default	Description
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.
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	[*OI only.*]Build a VM with the dynamic compiler.
CVM_JIT_USE_FP_HARDWARE	target-specific: see GNUmakefile	[*OI only.*] 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 not supported on the ARM port and will result in build errors.
CVM_JVMDI	false	Build a VM that supports Java debugging based on JVMDI. This option is not supported with CVM_JIT=true. When set true, there will be a significant degradation of performance.
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.

TABLE 5-1 Supported Makefile Options

Makefile Option	Default	Description
CVM_JVMPI_TRACE_INSTRUCTION	$(CVM_JVMPI)	Build a VM that supports bytecode tracing for profiling purposes. Enabling this option imposes a greater runtime burden on the interpreter. Hence, this option is provided in case the user does not need this feature and does not want the additional runtime burden to have an impact on the profile they are sampling.
CVM_MTASK	true if J2ME_PLATFORM=appmanager	Build with mtask enabled.
CVM_OPTIMIZED	!$(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.
CVM_PRELOAD_LIB	false	Build a VM with all the system and profile classes preloaded. See Chapter 9.
CVM_SYMBOLS	$(CVM_DEBUG)	Include debugging and symbol information for C code even if the build is optimized. Normaly, 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.

5.3.2 Limited Support Makefile Options

The makefile options described in TABLE 5-2 are limited in that their default values are supported, but alternate values are not and should be considered experimental.

TABLE 5-2 Limited Support Makefile Options

Makefile Option	Default	Description
`CVM_CCM_COLLECT_STATS`	`false`	[*OI only.*] Build a VM which collects statistics on the runtime activity of dynamically compiled code, even if the build is optimized.
`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_JIT_DEBUG`	`false`	[*OI only.*] Build the dynamic compiler with extra debugging support, including support for filtering which methods are compiled, and support for tracing the JCS rules used during compilation.
`CVM_JIT_PROFILE`	`false`	[*OI only.*] 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`	`$(CVM_JVMDI)`	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`.

TABLE 5-2 Limited Support Makefile Options

Makefile Option	Default	Description
CVM_TRACE_JIT	$(CVM_TRACE)	[*OI only.*] 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_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_XRUN	false	Build a VM which supports the -Xrun command-line option for loading native libraries. Defaults to true if either CVM_JVMDI or CVM_JVMPI are true.

CHAPTER 6

Debugging Support

Debugging is the exploration of the relationships between the source code structure of an application, the behavior of its compiled code and the capabilities of the target Java runtime environment. The CDC build system supports Java debugging through the Java Virtual Machine Debugger Interface (JVMDI). For more information about JVMDI, see `http://java.sun.com/j2se/1.4.2/docs/guide/jvmdi`.

This chapter describes how to enable JVMDI-based debugging in the CDC Java runtime environment. See the companion document *CDC Runtime Guide* for information about how to use a JVMDI-based debugger like `jdb` running on a Java SE host with a debugging-enabled CDC Java runtime environment running on a target device.

FIGURE 6-1 describes the Java Platform Debugger Architecture (JPDA) which allows a Java virtual machine to communicate with a Java debugging tool. Building the CDC Java runtime environment with JVMDI support inserts a Java Debug Wire Protocol (JDWP) agent in the Java runtime environment. It also allows the JDWP agent to communicate with a remote debugger.

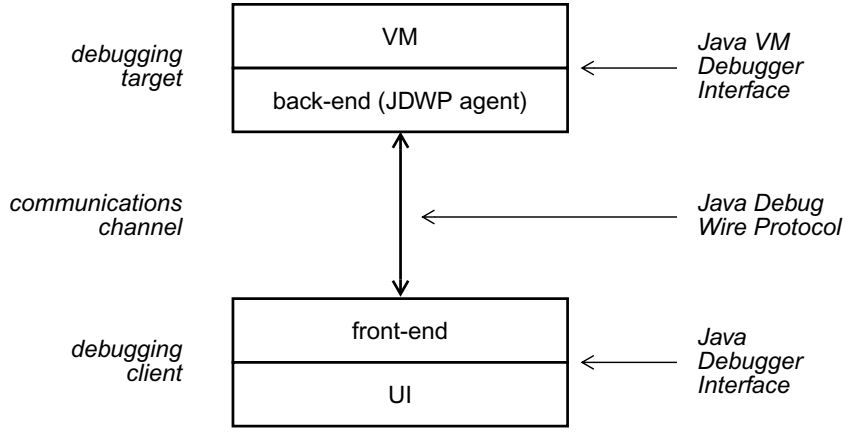

FIGURE 6-1 JVMDI Architecture

The source code for the JVMDI debugging support is located in the `share/tools/jpda` directory and the build files are located in `build/share`.

6.1 Building with JVMDI Support

The `CVM_JVMDI` makefile option enables debugging support in the CDC Java runtime environment.

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

The steps below demonstrate how to build a CDC Java runtime environment with Java debugging 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_JVMDI=true CVM_JIT=false
   ```

   This creates a VM executable in `bin/cvm` as well as the JDWP shared libraries in `jdwp/lib`.

3. **Make sure that the JDWP shared libraries are in the shared library search path so that the runtime shared library loader (`ld.so`(8)) can find them.**

   There are three ways to do this:

   - Move the shared libraries in the `jdwp/lib` directory into the `lib` directory:

     ```
     % mv jdwp/lib/* lib
     ```

   - Redefine the `LD_LIBRARY_PATH` environment variable to include the jdwp/lib directory.
   - Set the `java.library.path` system property on the `cvm` command-line.

4. **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 debugger running on a Java SE host system to a Java application running on a CDC Java runtime environment.

CHAPTER 7

Profiling Support

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.

7.1 Building with JVMPI Support

The `CVM_JVMPI` makefile 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 – The `CVM_JIT` option must be explicitly disabled for JVMPI builds if it is normally enabled by default.

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.

CHAPTER 8

Adding an Optional Package

The standard mechanism for extending the functionality of a CDC Java runtime environment is to add an optional package that provides technology-specific functionality. For example, the RMI optional package provides a distributed application programming model and the JDBC optional package provides access to different kinds of database systems.

This chapter describes how to add an optional package to the CDC build system, using the RMI optional package as an example. It focuses on build system procedures. The companion document *CDC Runtime Guide* describes the various forms of the Java class search path. The CDC Java runtime environment uses the *extensions class search path* to locate the classes in an optional package.

Note – In the Java ME platform, optional packages are based on profiles. Therefore, the CDC build system requires that an optional package be built for a specific CDC profile target like `foundation`, not for a lower-level build target like `cdc`.

8.1 Installing an Optional Package

The RMI optional package must be downloaded separately. For information about the RMI optional package and how to obtain it, see `http://java.sun.com/products/rmiop`.

The RMI optional package release notes include installation instructions for integrating the RMI optional package into the CDC build system. Basically, the Zip archive must be unbundled in the top-level directory. This will install the RMI source files into `src/share/rmi` and the build files into `build/share`.

8.2 Building an Optional Package

If the optional package follows the build and source file conventions described in Section 8.3, "Optional Package Makefile Naming Convention" on page 8-2, the CDC build system will detect it and build it when necessary.

8.2.1 Makefile Option Syntax

The `OPT_PKGS` makefile option indicates which optional packages are compiled with the current build. The syntax of this makefile option is:

`OPT_PKGS=<`*`pkg1`*`>[,<`*`pkg2`*`>] | all`

where <*pkg1*>, <*pkg2*> are the names of the optional packages. A comma is used, without spaces, to separate multiple optional package names. For example, the following command includes the RMI OP package in the Foundation Profile build:

```
% make J2ME_CLASSLIB=foundation OPT_PKGS=rmi
```

This produces the `lib/foundation_rmi.jar` file along with the other generated runtime files in the target build directory.

When `OPT_PKGS` is set to the value `all`, all the available optional packages are included in the build.

8.3 Optional Package Makefile Naming Convention

The naming convention for optional package makefiles is similar to the naming convention for profile makefiles in `build/share`. Each optional package must have a definitions file and a rules file. The naming convention for definitions and rules files is:

`defs_<`*`pkgname`*`>_pkg.mk`
`rules_<`*`pkgname`*`>_pkg.mk`

For example, for the RMI optional package these makefiles are:

```
defs_rmi_pkg.mk
rules_rmi_pkg.mk
```

The _pkg suffix alerts the build system that the file is an optional package makefile. When the OPT_PKGS build option is specified, the build process locates the optional package's makefiles and includes them in the build. For example, with the command:

```
% make J2ME_CLASSLIB=foundation OPT_PKGS=rmi,jdbc
```

the build process locates the files defs_rmi_pkg.mk, rules_rmi_pkg.mk, defs_jdbc_pkg.mk and rules_jdbc_pkg.mk and includes them in the build.

8.3.1 Optional Package Makefile Variables

Two variables must be included within each package's defs_<*pkgname*>_pkg.mk file. They are:

- OPT_PKGS_SRCPATH - the list of directories containing Java source files
- OPT_PKGS_CLASSES - the list of classes that make up the optional package

For example, if the optional package definitions makefile (defs_<*pkgname*>_pkg.mk) contains:

```
OPT_PKGS_SRCPATH += $(MY_ROOT)/src/mypackage/classes
OPT_PKGS_CLASSES += my.class.Class1 my.class.Class2
```

The += syntax is required when more than one optional package is added to the CDC build system.

8.3.2 javadoc Variables

Optional packages can include rules for generating javadoc API reference documentation for the optional package along with the profile's API documentation. Use the target javadoc.zip for generating the javadoc files.

The makefile variable OPT_PKGS_JAVADOC_RULES is the key to generating javadoc files for the optional package. Each makefile rule that added to OPT_PKGS_JAVADOC_RULES runs after the rules for the profile javadoc files are complete. For example, to add the RMI optional package to the set of javadoc build targets, the following rule can be added to rules_rmi_pkg.mk.

```
javadoc-rmi: $(INSTALLDIR)/javadoc-rmi $(JAVADOC_RMI_CLASSESLIST)
../../src/share/javadoc/rmi-overview-description.html
$(CVM_JAVADOC) options...
...
(cd $(INSTALLDIR); \
$(ZIP) -r -q - javadoc-rmi) > $(INSTALLDIR)/javadoc-rmi.zip
```

To include this rule with the current build, use the variable `OPT_PKGS_JAVADOC_RULES` in `defs_rmi_pkg.mk`. For example,

```
OPT_PKGS_JAVADOC_RULES += javadoc-rmi
```

The += syntax is required if more than one optional package is added to the CDC build system.

CHAPTER 9

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

9.1 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.

9.2 Enabling Preloaded Builds

The main makefile option for enabling preloaded builds is `CVM_PRELOAD_LIB`. When this option is set to `false`, the CDC build system will preload only a minimum set of system classes that the VM requires. When `CVM_PRELOAD_LIB` is set to `true`, the CDC build system will preload the entire CDC Java class library.

For example, the following make command builds a CDC Java runtime environment based on the Foundation Profile with Java class preloading enabled.

```
% make J2ME_CLASSLIB=foundation CVM_PRELOAD_LIB=true
```

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.

Note – Even if `CVM_PRELOAD_LIB` is set to `false`, the CDC build system still uses `JavaCodeCompact` to preload a certain number of system classes to simplify the task of launching the VM.

9.3 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.

`CVM_JCC_INPUT` specifies the list of preloaded classes. See Section A.4, "`JavaCodeCompact` Class Transitive Closure Requirements" on page A-4 for a description of the requirements for classes that can be preloaded by `JavaCodeCompact`.

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
   ```

   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, `CVM_JCC_INPUT` will preload only the classes appended to its definition. To preload the rest of the Java class library, `CVM_PRELOAD_LIB=true` and `J2ME_CLASSLIB=cdc` or `J2ME_CLASSLIB=foundation` are required.

   In this example, since `HelloWorld` is preloaded, it is not necessary to define the class search path with the `-cp` command-line option. 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**

`JavaCodeCompact` Reference

A.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.

A.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 A-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 A.5, "Output" on page A-4.
`-o` *outfilename*	Provides a template for the name of the output files to be produced. Section A.5, "Output" on page A-4.
`-qlossless`	Preserves more information about the original program in the output file for use of the debugging using the JVMDI debugger interface. See Section A.3, "Opcode Transformations" on page A-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 A-1 `JavaCodeCompact` Options

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 JVMDI 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.

A.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 JVMDI), 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.

A.4 `JavaCodeCompact` Class Transitive Closure Requirements

`JavaCodeCompact` determines the offset of every instance field and the method table offset of each non-static method for each class in the generated file. It requires that the complete inheritance hierarchy for each class be present in the set of preloaded classes. Also, all classes referenced from bytecodes must be preloaded because `JavaCodeCompact` requires full transitive closure of the preloaded classes.

A.5 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, giving forward declarations, and for use only by the other source files produced by `JavaCodeCompact`. It will be referred to hereafter as the forward file.

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.

A.6 See Also

- `java` - the Java application launcher
- `jar` - Java archive tool