



CLDC HotSpot™ Implementation Build Guide

CLDC HotSpot Implementation, Version 2.0
Java™ ME Platform

Sun Microsystems, Inc.
www.sun.com

May 2007

Copyright © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, HotSpot, J2ME, J2SE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, phoneME and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

The Adobe logo and the PostScript logo are trademarks or registered trademarks of Adobe Systems, Incorporated.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

CE PRODUIT CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L'AUTORISATION EXPRESSÉE, ECRITE ET PRÉALABLE DE SUN MICROSYSTEMS, INC.

Droits du gouvernement des États-Unis – logiciel commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut inclure des éléments développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, HotSpot, J2ME, J2SE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, phoneME et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et sous licence exclusive de X/Open Company, Ltd.

Intel est une marque déposée de Intel Corporation ou de sa filiale aux Etats-Unis et dans d'autres pays.

OpenGL est une marque déposée de Silicon Graphics, Inc.

Le logo Adobe et le logo PostScript sont des marques de fabrique ou des marques déposées de Adobe Systems, Incorporated.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DÉCLARATIONS ET GARANTIES EXPRESSÉS OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉ PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.



Contents

Preface xi

1. Introduction 1-1

1.1 Supported Platforms 1-1

1.2 Build Modes 1-2

1.3 Building in Batch Mode or with IDE 1-2

1.4 Developing for Linux Operating System 1-3

1.5 Developing in Microsoft Windows OS 1-3

1.6 Environment Variable Summary 1-4

1.7 Environment Variable Details 1-4

1.7.1 Forward and Backward Slashes 1-5

1.7.2 Spaces in File Paths 1-5

1.7.2.1 Discovering Abbreviated File Paths in Windows 1-5

1.7.3 Path Settings for Visual Studio 1-6

2. Required Compilers and Tools 2-1

2.1 Compiler Requirements 2-1

2.1.1 GNU Make 2-2

2.1.2 Obtaining MKS Toolkit 2-2

2.1.3 x86 Build Requirements 2-2

- 3. Directory Structure 3-1**
 - 3.1 Directory `build` 3-2
 - 3.2 Directory `src` 3-3
 - 3.3 Details of `vm/share` 3-4
 - 3.4 Details of `vm/cpu` 3-4
 - 3.5 Details of `vm/os` 3-7

- 4. Building and Running CLDC HotSpot Implementation 4-1**
 - 4.1 Getting Software to Compile 4-1
 - 4.2 Build Modes 4-2
 - 4.2.1 When Runtime Flags are Available 4-2
 - 4.3 Build Time Flags for Virtual Machine Configuration 4-3
 - 4.4 CLDC HotSpot Implementation Runtime Flags 4-10
 - 4.5 Running CLDC HotSpot Implementation 4-11
 - 4.5.1 Runtime Command Line Options 4-12
 - 4.5.2 Runtime Command Line Examples 4-13
 - 4.6 Categories of CLDC HotSpot Implementation Runtime Flags 4-14
 - 4.6.1 Product Mode Runtime Flags 4-14
 - 4.6.2 Development Mode Runtime Flags 4-17
 - 4.6.3 Conditional Mode Runtime Flags 4-24
 - 4.6.4 Tuning Flags for Performance 4-27

- 5. Building on Various Platforms 5-1**
 - 5.1 x86 Linux Platform, Batch Mode 5-1
 - 5.2 x86 Win32 Platform, Batch Mode 5-1
 - 5.3 x86 Win32 Platform, Batch Mode with CodeWarrior Compiler 5-2
 - 5.4 x86 Win32 Platform, IDE Mode 5-2
 - 5.5 ARM Linux Platform, Batch Mode 5-3
 - 5.6 ADS Linux Platform, Batch Mode 5-3

5.7	ADS Win32 Platform, Batch Mode	5-4
5.8	ARM Symbian Platform, Batch Mode	5-4
5.9	XScale Platform, Batch Mode	5-5
5.10	Building the Portable ROMizer	5-5
5.11	Building for Other Target Platforms	5-5
5.12	Building and Running Examples	5-6

Index Index-1

Tables

TABLE 1-1	Required Environment Variables	1-4
TABLE 3-1	Distribution Directories	3-1
TABLE 3-2	Directories in <code>build</code>	3-2
TABLE 3-3	Directories in <code>src</code>	3-3
TABLE 3-4	Directories in <code>src/vm/share</code>	3-4
TABLE 3-5	Files in <code>src/vm/cpu</code> - Generator	3-5
TABLE 3-6	Files in <code>src/vm/cpu</code> - Runtime System	3-5
TABLE 3-7	Files in <code>src/vm/cpu</code> - Compiler	3-6
TABLE 3-8	Files in <code>src/vm/os</code>	3-7
TABLE 4-1	CLDC HotSpot Implementation Build Modes	4-2
TABLE 4-2	Virtual Machine Build Time Configuration Flags	4-4
TABLE 4-3	Command Line Syntax for CLDC HotSpot Implementation Runtime Flags	4-10
TABLE 4-4	<code>cldc_hi</code> Command-line Options	4-12
TABLE 4-5	Product Mode Command Options	4-14
TABLE 4-6	Develop Mode Generate Options	4-17
TABLE 4-7	Develop Mode Runtime System Options	4-18
TABLE 4-8	Develop Mode Interpreter Options	4-19
TABLE 4-9	Develop Mode Compiler Options	4-21
TABLE 4-10	Develop Mode Object Heap Options	4-22
TABLE 4-11	Develop Mode Isolates Options	4-23

Preface

This document provides information for building the Connected Limited Device Configuration HotSpot™ Implementation virtual machine and libraries. CLDC HotSpot Implementation is a high-performance virtual machine that can be used as an execution engine for the Connected Limited Device Configuration platform.

Who Should Use This Document

This document is intended primarily for individuals and companies who want to port the CLDC HotSpot Implementation virtual machine to a new platform. It is also invaluable for implementation engineers who wish to implement an entire Java ME technology-based stack on top of the CLDC HotSpot Implementation virtual machine. The document is useful also to anyone who wants to learn more about the internal details of the CLDC HotSpot Implementation virtual machine.

How This Book Is Organized

This book has the following chapters:

[Chapter 1](#) describes the key design goals and the history of CLDC HotSpot Implementation.

[Chapter 2](#) describes the required compilers and other development tools for building and porting CLDC HotSpot Implementation.

[Chapter 3](#) describes the directory structure of the files in this release.

[Chapter 4](#) describes in detail how to build and run CLDC HotSpot Implementation.

[Chapter 5](#) describes the specific build instructions for the x86 version and the target platform that best suits your purpose.

Terminology

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

Java technology-based application	(Java application)
class contained in a Java class file	(Java class)
Java technology-based code	(Java code)
Java programming language debugger	(Java debugger)
Java programming language object heap	(Java heap)
Java technology level	(Java level)
Java technology-based object	(Java object)
Java technology-based packages	(Java packages)

Java programming language profiler	(Java profiler)
Java technology-based programs	(Java programs)
thread in a Java virtual machine representing a Java programming language thread	(Java thread)
stack used by a Java thread	(Java thread stack)

Related Documentation

The CLDC HotSpot™ Implementation Virtual Machine, A Technical White Paper, Sun Microsystems, Inc. (2005), which can be downloaded from <http://java.sun.com/javame/overview/techpapers/index.jsp/>

Connected, Limited Device Configuration Specification, Version 1.0, Java Community Process, Sun Microsystems, Inc. <http://www.jcp.org/en/jsr/detail?id=030>

Connected, Limited Device Configuration Specification, Version 1.1, Java Community Process, Sun Microsystems, Inc. <http://www.jcp.org/en/jsr/detail?id=139>

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

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)

Mobile Information Device Profile Specification, version 1.0, Java Community Process, Sun Microsystems, Inc. <http://www.jcp.org/en/jsr/detail?id=037>

Mobile Information Device Profile Specification, version 2.0, Java Community Process, Sun Microsystems, Inc. <http://www.jcp.org/en/jsr/detail?id=118>

The Java Hotspot™ Performance Engine Architecture, a technical white paper, Sun Microsystems, Inc. <http://java.sun.com/products/hotspot/whitepaper.html>

K Native Interface (KNI) Specification, (Sun Microsystems, Inc., 2002)

Programming Wireless Devices with the Java™ 2 Platform, Micro Edition by Roger Riggs, Antero Taivalsaari and Mark VandenBrink (Addison-Wesley 2001)

Programming Wireless Devices with the Java™ 2 Platform, Micro Edition, Second Edition, by Roger Riggs, Antero Taivalsaari, Jim Van Peurseem, Jyri Huopaniemi, Mark Patel, and Aleksi Uotila (Addison-Wesley 2003)

KVM Debug Wire Protocol (KDWP) Specification, (Sun Microsystems, Inc., 2002)

Introduction

Building and running CLDC HotSpot Implementation is a topic that goes beyond building only once. A porting effort for this technology will require building out of the box for study and evaluation, modifying the source code, makefiles and build environment to suit your product, and tuning and rebuilding the implementation numerous times. This manual describes the files contained in this release, tools requirements, and the many options and flags available.

This document explains how to build CLDC HotSpot Implementation in two different ways. You will typically be building:

The x86 version (under Win32, Linux or Symbian OS) and one of the supported target device platforms listed next.

1.1 Supported Platforms

The following platforms are directly supported by this release of CLDC HotSpot Implementation. You can modify the build environment if necessary to support your target platform.

- ARM processor (P2 Board) running MontaVista Linux (CEE 3.1 SDK developer tools)
- ARM 32-bit processor building and compiling with ADS tools
- ARM/Thumb processor building and compiling with ADS tools
- ARM/Thumb processor compiled with GCC
- ARM processor running Symbian OS
- ARM processor running proprietary OS

Note – The P2 Board is the Texas Instruments OMAP730 Software Development Platform.

Regardless of which is your target OS, if you have not previously built CLDC HotSpot, you will probably want to first build the x86 version. Refer to [Chapter 5](#) to find the build instructions for the x86 version and the target platform that best suits your purpose.

1.2 Build Modes

CLDC HotSpot can be built in the following three modes:

Debug: Includes all the possible debug and test code (including profiler and runtime assertions). This version executes rather slowly. The full complement of command line options is available.

Release: Another debug version that runs faster than the full **Debug** build because there are no runtime assertions (still includes profiler).

Product: The optimized product build. No debug or test code included. Only a subset of command line options is available.

1.3 Building in Batch Mode or with IDE

The standard and supported way to build CLDC HotSpot is as a batch process using the makefiles provided with this release. The makefile is multipurpose, and if environment variables are set correctly (refer to [Section 1.6, “Environment Variable Summary” on page 1-4](#)), simply typing `gnumake` will successfully build for the supported target platforms.

CLDC HotSpot can also be built in an IDE. For details, refer to [Section 5.4, “x86 Win32 Platform, IDE Mode” on page 5-2](#).

1.4 Developing for Linux Operating System

All Linux/x86 development is done with a GCC compiler.

Linux/ARM development requires either the specific GCC compiler for the ARM architecture, the ADS compiler (the Arm Development Suite), or a proprietary compiler provided by the hardware manufacturer of an ARM-based CPU.

1.5 Developing in Microsoft Windows OS

If developing on a Microsoft Windows workstation, set your environment variable `%PATH%` so that the path begins with UNIX-like tools (MKSNT), JDK utilities, GNU Make, and your C++ compiler.

The build procedures and makefiles require that you have GNU Make installed on your workstation. You will find a link to FTP sites for downloading GNU tools, including GNU Make, at the GNU Project web server, www.gnu.org:
<http://www.gnu.org/software/make/make.html>.

MKS Toolkit can be purchased online at:
<http://webstore.mksoftware.com/webstore/>.

If you are developing on a Microsoft Windows workstation, MKS Toolkit is required to support the batch mode build procedures.

For x86/Win32 development, Microsoft Visual C++ (Professional Version 6.0 or higher) or Visual Studio is a requirement. (Sometimes this compiler is referred to as MSVC++.)

The Microsoft Visual C++ compiler requires the variables `%LIB%` and `%INCLUDE%` to be defined correctly for batch builds. Any such variable consumed by the Windows operating system and Microsoft tools (and not by GNU Make) must use back slashes for file paths. This is true even when using the shell provided by MKS Toolkit. See the example batch files provided in the directory `misc/setup_examples`.

In addition to Microsoft Visual C++ (Visual Studio), x86/Win32 development requires the Processor Pack 5 for Visual C++, which provides the x86 macro assembler.

The Microsoft Macro Assembler must be version 6.15 or higher. (ML.exe version 6.15 is included in the current Microsoft Visual C++ Processor Pack. Note that the Macro Assembler is no longer required for Symbian builds on Microsoft Windows OS.)

1.6 Environment Variable Summary

You need to set the following environment variables:

TABLE 1-1 Required Environment Variables

Name	Required/Optional	Description
JVMWorkSpace	Required	Directory name of the CLDC HotSpot workspace. Must use forward slashes (/).
JVMBuildSpace	Optional	It specifies where the output files will be stored. If not set, the default value is \$(JVMWorkSpace)/build. Must use forward slashes (/).
INCLUDE	Required for win32	Point to the include directory inside your MSVC++ (for win32) installation.
LIB	Required for win32	Point to the lib directory inside your MSVC++ (for win32) installation.
PATH	Required	Must be set to include all compiler tools used in the build process.
JDK_DIR	Required	Must be set to point to the installation directory of your JDK.
X86_INCLUDE	Required for win32	Set it to be the same value as your INCLUDE variable for win32_i386 build.
X86_LIB	Required for win32	Set it to be the same value as your LIB variable for win32_i386 build.
X86_PATH	Required for win32	Set it to be the same value as your PATH variable for win32_i386 build.
ROMIZING	Optional	Default value is true.
SAVE_TEMPS	Optional	If set to true, the C++ compiler will save the intermediate .asm file it generates during the compilation of each .cpp file.

1.7 Environment Variable Details

This section covers important information about how the syntax of environment variables differs between platforms.

1.7.1 Forward and Backward Slashes

There is a difference in the file path notations of GNU Make and of the Microsoft Windows platform.

GNU Make assumes that directory paths contain forward slashes (/) rather than backward slashes (\). This is true whether the variable is passed at the command line or within a makefile. It's an important distinction when setting up environment variables. Variables such as *JVMWorkSpace* are consumed only by the `gnumake` command, and must use forward slashes. Environment variables such as `PATH` must use backward slashes, since this variable is interpreted by the host operating system.

1.7.2 Spaces in File Paths

There are several things that you need to know about the use of spaces in file paths:

- MSVC++ likes to be installed in directories that have spaces in their names (for example, `C:\Program Files`)
- MKS and `gnumake` don't particularly like spaces in file names or environment variables.

1.7.2.1 Discovering Abbreviated File Paths in Windows

Here's an exercise to get rid of spaces in your environment variables.

If, for example, your MSVC++ include directory is at
`C:\Program Files\Microsoft Visual Studio\VC98\include`

In a command window (running Windows NT or Windows 2000), type the following commands:

```
C:  
cd \  
dir /x
```

You will discover that `C:\Program Files` will now be referred to as something like `c:\PROGRA~1`. Using this designation, you can do a `cd` command to change directory to `C:\Program Files\Microsoft Visual Studio`, and you might find that this directory is referred to as something like `MICROS~4`. You can continue to repeat the same exercise to discover the short name (with ~) of any subdirectories whose name contains spaces or which is longer than 8 characters.

Finally you can set your `INCLUDE` environment variable to something like
`set INCLUDE=c:\PROGRA~1\MICROS~4\VC98\Include`

Note that the short names are different on each PC, so don't just cut and paste the above command!

Here's an example of the MSVC++ settings. You can save these in a batch file.

```
set PATH=c:\mksnt;F:\WINNT\system32;F:\WINNT;e:\jdk1.4\bin
set PATH=%PATH%;E:\Program Files\Microsoft Visual Studio\VC98\bin
set PATH=%PATH%;E:\Program Files\Microsoft Visual Studio\Common\
MSDev98\bin
set PATH=%PATH%;E:\Program Files\Microsoft Visual Studio\Common\
Tools\WinNT
set INCLUDE=E:\Program Files\Microsoft Visual Studio\VC98\include
set LIB=E:\Program Files\Microsoft Visual Studio\VC98\lib
```

You need to do some trial and error to determine the correct settings of *LIB*, *INCLUDE* and *PATH*:

1. Set some initial values for *LIB*, *INCLUDE* and *PATH* based on your best guess.
2. Run `gnumake`
3. Handle any errors that might be reported.

If you see an error about a missing `.exe` or `.dll`, find out where it is and add its path to *PATH*.

If you see an error about a missing `.h`, find out where it is and add its path to *INCLUDE*.

If you see an error about a missing `.lib`, find out where it is and add its path to *LIB*.

4. Go back to step 1 and try again.

1.7.3 Path Settings for Visual Studio

Correct *PATH* settings are required by the IDE tools such as Microsoft Visual Studio. Normally these are set in your registry when installing these tools. However, there have been cases where the installers don't set them correctly. Use the `path` command at the command line to verify these path settings. You should see something like:

```
C:\Program Files\Microsoft Visual Studio\Common\Tools\WinNT;
C:\Program Files\WinNT;
C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;
C:\Program Files\Microsoft Visual Studio\Common\Tools;
C:\Program Files\Microsoft Visual Studio\VC98\bin;
C:\jdk1.4\bin
```

Required Compilers and Tools

The requirements for building and porting the implementation of CLDC HotSpot Implementation include compiler requirements and other development tool requirements.

Note – The tool requirements for building the implementation of CLDC HotSpot Implementation provided in this release might differ from those for building the ported implementation on your target platform.

To use the CLDC HotSpot™ Implementation build system, you need the following software:

- J2SE Development Kit (JDK) version 1.4.1_07
- Microsoft Visual C++ (Professional Version 6.0 or higher) or Visual Studio.
- GNU Make (gnumake), and for some platforms the GNU C++ compiler.
- MKS Toolkit (for building on Microsoft Windows host workstations).

See [Section 1.5, “Developing in Microsoft Windows OS” on page 1-3 OS](#) for information on obtaining GNU Make and MKS Toolkit.

The following software is required only if you are developing in the corresponding environments:

- If using the ADS tools for ARM, version 1.2 or higher is required.
- If using the Symbian SDK, version 7.0 is required.

2.1 Compiler Requirements

All make files provided with this release require GNU Make, version 3.79.1 or later.

Building CLDC HotSpot Implementation on the ARM platform requires version 3.0 or later of Microsoft eMbedded Visual C++.

If using the ADS tools for ARM, version 1.2 or later is required.

2.1.1 GNU Make

The build procedures require that you have GNU Make installed on your workstation. For workstations running Microsoft Windows, GNU Make is available as part of the MKS toolkit. If you need to update the version of GNU Make, you will find a link to FTP sites for downloading GNU tools, including GNU Make, at the GNU Project web server, www.gnu.org:

<http://www.gnu.org/software/make/make.html>

2.1.2 Obtaining MKS Toolkit

MKS Toolkit offers the convenience of UNIX-like tools for developing on Microsoft Windows workstations. In building the CLDC HotSpot Implementation system, the MKS tools work better than alternatives such as Cygwin tools. MKS Toolkit can be purchased online at:

<http://webstore.mkssoftware.com/webstore/>

2.1.3 x86 Build Requirements

For ports targeted to the x86 platform, building and developing requires Microsoft Visual C++ (Professional Version 6.0 or higher) or Visual Studio. Also required is the Processor Pack 5 for Visual C++, which provides the x86 macro assembler. For users of Visual C++ or Visual Studio, this is a free download from Microsoft:

<http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.aspx>

Note – After installing the Processor Pack, you can confirm the version of the x86 macro assembler by starting a command prompt and typing `m1.exe` without arguments. Your version number should be reported as Microsoft (R) Macro Assembler Version 6.15.xxxx where xxxx is the build number.

Any compiler used to generate code for the ARM processor must support 64-bit operations. (This limitation might be removed in a future version of CLDC HotSpot Implementation.)

Directory Structure

This release is provided as a single zip file. Unzip it into a convenient location on your workstation. This will result in a file tree with the parent directory named `cldc`.

All the directories and paths given in this chapter are contained in this parent directory. The main directories contained in the parent are:

- `build`
- `doc`
- `src`
- `bin`

The contents of these directories are detailed in [TABLE 3-1](#).

TABLE 3-1 Distribution Directories

Directory	Description
<code>build</code>	Contains makefiles for building debug and production versions of CLDC HotSpot Implementation on various CPUs and development platforms.
<code>doc</code>	Contains all documentation. (The file <code>cldc/index.html</code> is an index to documentation.)
<code>src</code>	Contains the C++ source code for the CLDC HotSpot Implementation virtual machine and the CLDC libraries. Also contains Java technology-based code (Java code) for tools for building and porting.
<code>bin</code>	Contains pre-compiled binary executables of the CLDC HotSpot Implementation virtual machine for various platforms. Also contains pre-built classes in archives under <code><platform>/dist/lib/cldc_classes.zip</code>

3.1 Directory build

TABLE 3-2 gives an overview of the directories of makefiles contained in build.

TABLE 3-2 Directories in build

Directory	Description
win32_i386	Contains a master GNU makefile and a .cfg (configuration) file for building CLDC HotSpot Implementation for the x86 processor under the Microsoft NT/98/2000 (Win32) operating systems. This includes building debug and production versions of CLDC HotSpot Implementation.
win32_i386_ide	Contains a batch file for creating an IDE project under Win32 OS.
linux_i386	Contains a master GNU makefile and a .cfg (configuration) file for building CLDC HotSpot Implementation for the x86 processor under Linux. This includes creating an IDE project and building debug and production versions of CLDC HotSpot Implementation.
linux_arm	Contains a master GNU makefile and a .cfg (configuration) file for building CLDC HotSpot Implementation for the ARM processor under Linux. This includes creating an IDE project and building debug and production versions of CLDC HotSpot Implementation.
ads_arm	For little-endian ARMulator platform.
ads_bigarm	For big-endian ARMulator platform.
ads_jazelle	For building the ARM Jazelle port with ADS tools.
linux_jazelle	For building the ARM Jazelle port under Linux OS.
linux_c	For building the portable ROMizer from C-language sources under Linux OS.
win32_c	For building the portable ROMizer from C-language sources under Win 32 OS.
solaris_c	For building the portable ROMizer from C-language sources under Solaris.
linux_bvd	For building the port of the XScale processor under Linux OS.
share	For each platform, contains IDE project files for creating the build space and workspace, and executables for the preverifier, JAM, and other tools.

3.2 Directory `src`

TABLE 3-3 gives an overview of the directories of the C++ source and header files contained in `src`.

TABLE 3-3 Directories in `src`

Directory	Description
<code>vm</code>	Contains the source and header files that pertain to the core virtual machine. Also includes the <code>includeDB</code> and ROMized CLDC library files.
<code>vm/share</code>	Contains the virtual machine source and header files that are shared by all ports, independent of any particular operating system or CPU architecture.
<code>vm/cpu/i386</code>	Contains the virtual machine source and header files that pertain to the x86 variants of the CLDC HotSpot Implementation.
<code>vm/cpu/arm</code>	Contains the virtual machine source and header files that pertain to the ARM processor variants of the CLDC HotSpot Implementation.
<code>vm/cpu/c</code>	Contains the source and header files that pertain to the portable C-language ROMizer.
<code>vm/os</code>	Contains the source and header files that pertain to the operating system-specific parts of the virtual machine.
<code>vm/os/win32</code>	Contains the source and header files that pertain to the implementation of the virtual machine on the Microsoft Windows NT/98/2000 operating system.
<code>vm/os/linux</code>	Contains the source and header files that pertain to the implementation of the virtual machine on the linux operating system.
<code>vm/os/ads</code>	Contains the source and header files that pertain to the implementation of the virtual machine using the ARM Developer's Suite.
<code>tools</code>	Contains various tools used by the build process.
<code>javaapi</code>	Contains the source code of the Java ME CLDC 1.0 and CLDC 1.1 libraries that are provided with the virtual machine.
<code>anilib</code>	Contains the source and header files that implement the functions in the Asynchronous Native Interface (ANI) for multiple threading.

3.3 Details of `vm/share`

TABLE 3-4 gives an overview of the directories of the C++ source and header files contained in `src/vm/share`. This code is intended to be common across all ports.

TABLE 3-4 Directories in `src/vm/share`

Directory	Description
<code>handles</code>	Handles provide GC safe references to objects. This directory contains the source and header files that pertain to handle creation and management in the core virtual machine. Contains abstractions for accessing heap objects inside the virtual machine.
<code>memory</code>	Contains the source and header files that pertain to the garbage collector and the memory system in the core virtual machine.
<code>runtime</code>	Contains the source and header files that pertain to runtime operations in the core virtual machine. This includes the code for stack frames, threads, monitors, scheduling, etc.
<code>utilities</code>	Contains the source and header files that pertain to utilities used by the other modules to implement virtual machine functionality.
<code>natives</code>	Contains native methods and interfaces implementing native functions.
<code>interpreter</code>	Contains the source and header files that implement the byte code interpreter.
<code>compiler</code>	Contains the source and header files that implement the adaptive compiler.
<code>debugger</code>	Contains the source and header files that implement the debugger interface for Java code.
<code>float</code>	Contains the source and header files that implement floating point.
<code>ROM</code>	Contains the source and header files that implement the ROMizer.
<code>verifier</code>	Contains the source and header files that implement the class file verifier.

3.4 Details of `vm/cpu`

The following tables give details of the CPU-specific C++ source and header files contained in `src/vm/cpu`. The files in this directory are specific to a certain CPU architecture, and they are related to three logical components: the interpreter generator, runtime system and the adaptive compiler.

Generator

The following files located in `src/vm/cpu/<cpu>` are related to the interpreter generator. The generator produces the machine-specific interpreter code (file `Interpreter_<cpu>.asm`). The generator is only present in a non-product build.

TABLE 3-5 Files in `src/vm/cpu` - Generator

File	Description
<code>Assembler_<cpu>.{hpp,cpp}</code>	Base class for <code>SourceAssembler</code> and <code>BinaryAssembler</code> .
<code>SourceAssembler_<cpu>.{hpp,cpp}</code>	Component for generating interpreter loop CPU instructions.
<code>SourceMacros_<cpu>.{hpp,cpp}</code>	Component for generating common idioms like <code>get_thread()</code> .
<code>Disassembler_<cpu>.{hpp,cpp}</code>	Disassembler used on arm to generate textual instructions from binary code. (Used for debugging.)
<code>TemplateTable_<cpu>.cpp</code>	Code for interpreting the different byte codes.
<code>InterpreterGenerator_<cpu>.cpp</code>	Code for <code>method_entry</code> , etc.
<code>NativeGenerator_<cpu>.cpp</code>	Hand-coded native methods like <code>java.lang.System.arraycopy</code> .
<code>SharedStubs_<cpu>.cpp</code>	Stubs used by both interpreter and compiled code.
<code>InterpreterStubs_<cpu>.cpp</code>	Stubs used by interpreter code.
<code>CompilerStubs_<cpu>.cpp</code>	Stubs used by compiled code.

Runtime System

The following files in `src/vm/cpu/<cpu>` implement important CPU-specific parts of the runtime system:

TABLE 3-6 Files in `src/vm/cpu` - Runtime System

File	Description
<code>GlobalDefinitions_<cpu>.hpp</code>	CPU-specific global definitions.
<code>GlobalDefinitions_<compiler>.hpp</code>	Compiler-specific global definitions. These files are part of the CLDC HotSpot Implementation porting interface.
<code>Bytes_<cpu>.hpp</code>	Tells how to read bytes, shorts and ints, either in native code or Java code.

TABLE 3-6 Files in `src/vm/cpu` - Runtime System

File	Description
<code>ObjectHeap_<cpu>.hpp</code>	Fast bit operations for the bit marking array.
<code>InterpreterRuntime_<cpu>.{hpp,cpp}</code>	Runtime interpreter support.
<code>Frame_<cpu>.{hpp,cpp}</code>	CPU-specific code for manipulating an execution activation.
<code>Debug_<cpu>.cpp</code>	(Used only in debug or release builds.) Code for printing the stack during debugging sessions.
<code>CompiledMethod_<cpu>.cpp</code>	(Used only in debug or release builds.) Helper functions to enable disassembling of the generated code on the fly.
<code>CompiledMethodDesc_<cpu>.cpp</code>	Contains CPU-specific relocation behavior and LRU patching behavior.
<code>kni_md.h</code>	KNI native function interface.

Compiler

The following files in `src/vm/cpu/<cpu>` implement the CPU-specific parts of the adaptive compiler:

TABLE 3-7 Files in `src/vm/cpu` - Compiler

File	Description
<code>Addressing_<cpu>.{hpp,cpp}</code>	Handles address and offset computations for the compiler.
<code>BinaryAssembler_<cpu>.{hpp,cpp}</code>	Interface for generating code into a <code>CompiledMethod</code> .
<code>Instructions_<cpu>.{hpp,cpp}</code>	Abstractions for decoding instructions used during compilation.
<code>CodeGenerator_<cpu>.{hpp,cpp}</code>	CPU-specific part of the code generator.
<code>CodeOptimizer_<cpu>.{hpp,cpp}</code>	Code optimization algorithms.
<code>CompilationQueue_<cpu>.{hpp,cpp}</code>	Compilation queue algorithms.
<code>FloatSupport_<cpu>.{hpp,cpp}</code>	Floating point and trigonometric algorithms.
<code>RegisterAllocator_<cpu>.cpp</code>	Register mapping for integer registers.
<code>FPURegisterMap_<cpu>.{hpp,cpp}</code>	Register mapping for FPU registers. (Note that the CLDC 1.0 Specification does not support floating point. This support is built in for a future version of the CLDC Specification.)

3.5 Details of `vm/os`

The following C++ source and header files in directory `src/vm/os/<os_family>` implement the operating system (OS) specific parts of the virtual machine:

TABLE 3-8 Files in `src/vm/os`

File	Description
<code>JVM_<os_family>.cpp</code>	Contains the main entry point for CLDC HotSpot Implementation, and code for handling incoming parameters when the virtual machine is launched from the command line.
<code>Os_<os_family>.cpp</code>	Platform-dependent implementation of the operating system abstraction. The functions in this file serve as “glue” to the operating system.
<code>OsFile_<os_family>.hpp</code>	Interface to file system calls on target OS.
<code>OsMemory_<os_family>.hpp</code>	Interface to memory management on target OS.
<code>OsMisc_<os_family>.hpp</code>	Interface to miscellaneous system calls on target OS.
<code>OsSocket_<os_family>.cpp</code>	Interface to sockets calls on target OS.
<code>Globals_<os_family>.hpp</code>	Target-OS-specific differences in command line switches and compile-time configuration options.

Building and Running CLDC HotSpot Implementation

This chapter describes the general steps to build CLDC HotSpot Implementation as released. Understanding how to build and run the system is a preliminary to beginning the porting effort.

This chapter also details how to run the CLDC HotSpot Implementation executable after it is built.

Refer to [Chapter 5](#) to find the specific build instructions for the x86 version and the target platform that best suits your purpose.

4.1 Getting Software to Compile

The first step upon first obtaining CLDC HotSpot Implementation is to build the x86 version. Although prebuilt executables are provided with this release, any porting effort will require numerous rebuilds of the system after modification of various parts of the source code. It is essential that the porting engineer get practice building the system with a variety of options and in the different “build modes” (described in the next section.)

Important cautions and advice for building are given in this section. Detailed step by step procedures are given in the file:

`doc/CDLC_HI-HowToBuild.html`.

These procedures include instructions for building debuggable versions of CLDC HotSpot Implementation executable (`cldc_hi.exe`), as well as optimized versions that include no debugging hooks. There are separate instructions for building within

an IDE (such as Microsoft Visual Studio) and for building from the command prompt (in batch mode). You can build the x86 version of CLDC HotSpot Implementation under Linux, which requires no Microsoft tools.

After building the CLDC HotSpot Implementation system, follow the instructions in [Section 4.5, “Running CLDC HotSpot Implementation” on page 4-11](#) to run it.

4.2 Build Modes

The CLDC HotSpot Implementation system can be built in three different modes:

- *Debug*: Includes all the possible assertions, debug and test code. This build version executes relatively slowly, but is very valuable for testing and debugging the system.
- *Release*: A version for development that runs significantly faster than the *debug* build. It includes few runtime assertions. It has no debug symbols, but supports all development command line options.
- *Product*: The optimized product build. This build version includes no testing or debug code or runtime assertions, and results in the most compact executable. It supports fewer command line options.

The differences between the main build modes are further detailed in this table:

TABLE 4-1 CLDC HotSpot Implementation Build Modes

Debug	All runtime assertions	Profiling	All command line options available	Runtime checks for which features enabled
Release	Few runtime assertions	Profiling	All command line options available	Runtime checks for features like ROMizing enabled
Product	No runtime assertions	No profiling	Fewer command options available	No runtime feature checks

An example of a possible runtime assertion is checking on each bytecode dispatch.

4.2.1 When Runtime Flags are Available

A *develop* build flag is available if the system is built in either of debug or release modes. A *product* build flag is available if the system is built in product mode. A *conditional* build flag is one that may or may not be available at runtime, depending on whether the virtual machine is built *with* or *without* in-place execution support.

To determine when a particular flag is available, see the relevant #define statements in `src/vm/share/utilities/`.

If a flag is declared as `develop` in `Globals.hpp`, it is available only in *debug* and *release* modes, but not in *product* mode.

If a flag is declared as `product` in `Globals.hpp`, it is available in all of *debug*, *release*, and *product* modes.

If a flag is declared as `optional` in `Globals.hpp`, it is called *conditional*. it behaves as:

- a *develop* flag if in-place execution *is not* enabled
- a *product* flag if in-place execution *is* enabled

If an conditional build mode flag is considered *product* according to the above rules, it could be modified at runtime in a virtual machine build as *product*. If the flag does not meet the rule, it is considered *develop* and cannot be modified at runtime in a virtual machine build as *product*. To see the rule for a particular conditional flag, see the relevant #define statements in `src/vm/share/utilities/Globals.hpp`.

Beginning in [Section 4.6, “Categories of CLDC HotSpot Implementation Runtime Flags” on page 4-14](#), the flags and options of CLDC HotSpot Implementation are categorized according to whether they are always available (*product* mode); available only if the system is built in either of *debug* or *release* modes; or conditionally available as just defined.

4.3 Build Time Flags for Virtual Machine Configuration

The set of flags named in the form `ENABLE_XXX` are used at build time to control the configuration of the virtual machine. These flags can be controlled directly or indirectly without editing code. The value of each of these flags is either 0 (*false*) or 1 (*true*). Unless otherwise specified in [TABLE 4-2](#), the default is *false*. However, the default value can differ between *product* mode and non-*product* mode builds.

The defaults for these flags are preset in the file `src/vm/share/utilities/BuildFlags.hpp`. They can be overridden from a user's environment variable or from a platform configuration (`.cfg`) file.

The program `src/tools/buildtool/BuildTool.java` is run at build time to create the file `jvmconfig.h`. For each flag, the program looks in the following sequence to determine the effective value.

1. environment variables

2. settings from the relevant platform configuration file (such as `build/linux_i386/linux_i386.cfg`)
3. the defaults set in `BuildFlags.hpp`

For example, if the default value of a flag is changed in the platform configuration file, that override becomes the effective setting, regardless of what is coded as the default in `BuildFlags.hpp`.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
<code>ENABLE_APPENDED_CALLINFO</code>	Compiler-specific. Append all call info records for a method at the end of compiled code of that method.
<code>ENABLE_ARM_V5TE</code>	Support instructions in the ARMv5TE architecture (such as BLX or CLZ).
<code>ENABLE_ARM_V6</code>	Support instructions in the ARMv6 architecture.
<code>ENABLE_ARM_V6T2</code>	Support instructions in the ARMv6T2 architecture (Thumb-2).
<code>ENABLE_ARM_V7</code>	Support instructions in the ARMv7 architecture.
<code>ENABLE_ARM_VFP</code>	Support ARM VFP instructions.
<code>ENABLE_ARM9_VFP_BUG_WORKAROUND</code>	Work around ARM9+VFP hardware feature.
<code>ENABLE_BRUTE_FORCE_ICACHE_FLUSH</code>	Generate brute-force code to flush the instruction cache (for platforms without OS support).
<code>ENABLE_BYTECODE_FLUSHING</code>	Special code needed for bytecode rewriting.
<code>ENABLE_CLDC_11</code>	Support CLDC 1.1 Specification instead of CLDC 1.0. (default = true.)
<code>ENABLE_C_INTERPRETER</code>	Use Java bytecode interpreter written in C code instead of the generated assembler interpreter.
<code>ENABLE_CODE_OPTIMIZER</code>	Enable optimization of code generated by dynamic compiler for a specific CPU.
<code>ENABLE_CODE_PATCHING</code>	Use code patching mechanism for timer tick checking optimizations.
<code>ENABLE_COMPILER</code>	Add the dynamic adaptive compiler for byte code execution. (default = true.)
<code>ENABLE_COMPILER_TYPE_INFO</code>	Maintain object type information during compilation.
<code>ENABLE_COMPRESSED_VSF</code>	Compiler-specific. Include table of compressed VSF in the Relocation of CompiledMethod to reduce the produced code size.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_CPU_VARIANT	Enable specialized features for a variant of the main CPU type.
ENABLE_CSE	Eliminate memory access related common byte code
ENABLE_DETAILED_PERFORMANCE_COUNTERS	Enable fine-grained performance counters. These counters may skew execution time.
ENABLE_DISPATCH_TABLE_ALIGNMENT	Make dispatch table 1024 aligned.
ENABLE_DISPATCH_TABLE_PADDING	Add extra entries to the dispatch table.
ENABLE_DYNAMIC_NATIVE_METHODS	Add ability to execute user classes containing native methods.
ENABLE_DYNAMIC_RESTRICTED_PACKAGE	Allow restricted packages to be dynamically specified.
ENABLE_EMBEDDED_CALLINFO	Compiler-specific. Embed call info records in compiled code just after the call instruction. This is used to debug the correctness of ENABLE_APPENDED_CALLINFO.
ENABLE_FAST_CRC32	Use fast CRC32 routine. (Adds 1KB footprint.) (default = true.)
ENABLE_FAST_MEM_ROUTINES	Use built-in memcmp and memcpy routines in the generated interpreter loop.
ENABLE_FLOAT	Support floating point byte codes. (default, true.)
ENABLE_FULL_STACK	Jazelle only. Use Full Java Stack. (default = true.)
ENABLE_HARDWARE_TIMER_FOR_TICKS	Nucleus-XScale only. Include code to set up a hardware timer to provide timer ticks to Nucleus. Even when enabled, the timer is still controllable with generic runtime flags. See Globals_nucleus.hpp.
ENABLE_HEAP_NEARS_IN_HEAP	Ensure all nears of romized HEAP objects and prototypical nears of all classes are in ROM HEAP block by cloning those nears from ROM TEXT and DATA blocks to the ROM HEAP block. Speeds up GC, but slightly increases footprint.
ENABLE_INCREASING_JAVA_STACK	If true, the Java stack grows upwards. That is, when an item is pushed, the stack pointer's value increases.
ENABLE_INLINE	Inline simple methods into their callers in compiled code.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_INLINEASM_INTERPRETER	If true, the interpreter loop is generated as a C file with inlined assembly code. This option is used on x86 only.
ENABLE_INLINE_COMPILER_STUBS	Compiler-specific. Generate inlined code for creating new objects and type arrays (instead of calling <code>compiler_new_object</code> and <code>compiler_new_type_array</code> stubs).
ENABLE_INLINED_ARRAYCOPY	Inline <code>arraycopy()</code> calls in compiled code.
ENABLE_INTERNAL_CODE_OPTIMIZER	Improved code optimizer for scheduling ARM instructions.
ENABLE_INTERPRETATION_LOG	Use a log of the most recently interpreted methods to make sure hot methods are compiled. (True but disable when running on slow devices.)
ENABLE_INTERPRETER_GENERATOR	Include code for generating interpreter loop in non-PRODUCT build.
ENABLE_ISOLATES	Add Isolate support to the virtual machine.
ENABLE_JAR_ENTRY_CACHE	Cache the JAR entry table for fast lookup.
ENABLE_JAR_READER_EXPORTS	Export routines for the JAR reader.
ENABLE_JAVA_DEBUGGER	Add Java debugger support.
ENABLE_JAVA_DEBUGGER_OLD_JAVAC	Add special code in method entry to clear locals if code compiled with old compiler.
ENABLE_JAVA_STACK_TAGS	Interleave values on the Java stack with tags.
ENABLE_JAZELLE	Enable support for Jazelle(TM) hardware acceleration of Java bytecode execution. This is a meta flag that enables a number of other build flags.
ENABLE_JVMPI_PROFILE	To support JVMPI profiler.
ENABLE_JVMPI_PROFILE_VERIFY	To support JVMPI profiler verification.
ENABLE_KVM_COMPAT	Support the <code>kvmcompat</code> module, which provides limited compatibility with KVM-based (pre-KNI) native methods.
ENABLE_LOOP_OPTIMIZATION	Simplify the code sequence at end of a loop.
ENABLE_MEASURE_NATIVE_STACK	Measures the amount of native stack the virtual machine uses.
ENABLE_MEMORY_MAPPED_FILES	Use memory-mapped files for loading binary images. This flag takes effect only if the target platform has <code>SUPPORTS_MEMORY_MAPPED_FILES=1</code> .

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_MEMORY_PROFILER	Add Memory Profiler support.
ENABLE_METHOD_TRAPS	Add MethodTrap= command-line option for special handling of a particular Java method invocation. Used for startup time measurements.
ENABLE_MINIMAL_ASSERT_BUILD	Turn off all non-product options unrelated to runtime assertion checks. This allows you to build a smaller Debug mode VM that still has assertions checks.
ENABLE_MONET	Enable on-device support (conversion and loading) of binary application image files for fast class loading.
ENABLE_MONET_COMPILATION	Enable on-device method precompilation. Requires ENABLE_MONET.
ENABLE_MONET_DEBUG_DUMP	Create debug dump files that describe the contents of binary ROM image files.
ENABLE_MULTIPLE_PROFILES_SUPPORT	Add support for using multiple profiles that may provide mutually exclusive APIs. Allows the hiding of certain classes under a specific profile.
ENABLE_NATIVE_ORDER_REWRITING	Enable rewriting of various bytecodes so their fields are in native ordering. (default = true.)
ENABLE_NPCE	Null-pointer check elimination. This requires OS support for exceptions when accessing address 0x0.
ENABLE_OOP_TAG	Support for debug int in oopdesc used for MVM GC tracing.
ENABLE_PAGE_PROTECTION	Use the mechanism of protected memory pages for certain compiler optimizations (e.g. check_timer_tick). Works only if the feature is supported by OS.
ENABLE_PCSSL	Enable support for the Portable Common Services Library.
ENABLE_PERFORMANCE_COUNTERS	Implement various performance counters.
ENABLE_PREINITED_TASK_MIRRORS	Put TaskMirror to a separate section of SystemROM image to allow loading them during startup of each task.
ENABLE_PRODUCT_PRINT_STACK	Include the debug function pss() in product build (useful for diagnosing deadlocks).
ENABLE_PROFILER	Add (Java) profiling support.
ENABLE_REFLECTION	Add Reflection support.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_REMEMBER_ARRAY_CHECK	Remember the length checking result of a unchanged local variable.
ENABLE_REMEMBER_ARRAY_LENGTH	Remember the length of the last accessed array in a register.
ENABLE_REMOTE_TRACER	Add remote tracing capabilities.
ENABLE_ROM_DEBUG_SYMBOLS	Use extra symbols in ROMImage.cpp to aid debugging (for example, recover original name of renamed classes).
ENABLE_ROM_JAVA_DEBUGGER	Add Java debugger support for ROMized. classes.
ENABLE_ROM_GENERATOR	Include code for generating (source or binary) ROM image.
ENABLE_RVDS	Support for ARM RealView Developer Suite (2.0 or later).
ENABLE_SEGMENTED_ROM_TEXT_BLOCK	Split the TEXT block of a source ROM image into several segments.
ENABLE_SEMAPHORE	Include <code>com.sun.cldc.util.Semaphore</code> class.
ENABLE_SOFT_FLOAT	Use the software floating point operations.
ENABLE_SOURCE_GENERATORS	Include code for generating interpreter loop and ROM image in non-PRODUCT build. (default = true in non-PRODUCT build.)
ENABLE_STACK_TRACE	Include code for printing the stack trace of Java Throwable objects. (default = true. Always enabled in non-PRODUCT builds.)
ENABLE_STATIC_TRAMPOLINE	Use static trampoline in dispatch to subroutine instead.
ENABLE_SYSTEM_CLASSES_DEBUG	Build system classes with <code>-g</code> and <code>+MakeROMDebuggable</code> . Must be passed in via make command line.
ENABLE_SYSTEM_ROM_HEADER_SKIPPING	Skip object headers in the system ROM image to save space.
ENABLE_SYSTEM_ROM_OVERRIDE	Allow a debug-mode VM to override the system ROM image by loading system classes from <code>classes.zip</code> .
ENABLE_THUMB_COMPILER	Generate compiled Java methods in THUMB mode
ENABLE_THUMB_GP_TABLE	Generate inverted GP table. Allows for small GP register offsets to aid the THUMB compiler.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_THUMB_LIBC_GLUE	Linux-only: Use glue code inside the VM for invoking functions in the GNU LIBC. Use this option if the VM is built in THUMB mode but your LIBC is not built with interworking.
ENABLE_THUMB_REGISTER_MAPPING	Generate ARM interpreter with fp and jsp as low registers. Critical for THUMB compiler.
ENABLE_THUMB_VM	Enable THUMB interworking in interpreter and compiled Java methods, so that the native code in the VM can be compiled in THUMB mode.
ENABLE_THUMB_COMPILER	Generate compiled Java methods in THUMB mode.
ENABLE_THUMB_REGISTER_MAPPING	Generate ARM interpreter with fp and jsp as low registers. Critical for THUMB compiler.
ENABLE_THUMB_GP_TABLE	Generate inverted GP table. Allows for small GP register offsets to aid the THUMB compiler.
ENABLE_TIMER_THREAD	Use a thread to generate timer ticks instead of a timer signal handler. (default = true.)
ENABLE_TOS_CACHING	Jazelle only. Allow TOS caching. (default = true.)
ENABLE_TRAMPOLINE	Use branch instruction to replace mov pc, rc in static method invoking. This is faster on the Xscale which has branch prediction.
ENABLE_TTY_TRACE	Enable the various TraceXXX flags.
ENABLE_VERBOSE_ASSERTION	Print detailed error messages when a run-time assertion fails.
ENABLE_VERIFY_ONLY	Add support for using the virtual machine as a tool for classpath verification without any byte code execution.
ENABLE_VM_MIPS	Enable measuring the speed of ARM CPU. Requires ENABLE_PERFORMANCE_COUNTERS.
ENABLE_WTK_PROFILER	Add WTK-compatible profiling support.
ENABLE_XSCALE_PMU_CYCLE_COUNTER	Use the PMU cycle counter on Intel Xscale CPU for performance measurement.
ENABLE_XSCALE_WMMX_ARRAYCOPY	Use XScale WMMX to implement System.arraycopy().
ENABLE_XSCALE_WMMX_INSTRUCTIONS	Use XScale WMMX instructions in compiled code and interpreter.

TABLE 4-2 Virtual Machine Build Time Configuration Flags

Flag	Description
ENABLE_XSCALE_WMMX_TIMER_TICK	Use XScale WMMX registers to check for timer ticks.
ENABLE_ZERO_YOUNG_GENERATION	Fills young generation with zero values after garbage collection. When the option is off, each newly created object is cleared right after allocation.
ROMIZING	Set this environment variable to <i>true</i> to build a virtual machine that has romized system classes. (The default is <i>false</i> .)

4.4 CLDC HotSpot Implementation Runtime Flags

The CLDC HotSpot Implementation system has a special mechanism that allows most of the runtime options of the system to be treated uniformly either by setting them in `Globals.hpp` or at the command line. These options are called *runtime flags*. The system is implemented so that the names and functionality of the flags in `Globals.hpp` are the same as the runtime command-line options. The command-line syntax for runtime flags is as follows:

TABLE 4-3 Command Line Syntax for CLDC HotSpot Implementation Runtime Flags

+<option>	force option to <code>true</code>
-<option>	force option to <code>false</code>
=<option><value>	set numeric option to <value>. (There are no spaces in the expression.)

The syntax for invoking the equivalent functionality as compilation flags or runtime command-line options is slightly different, though the names of the options are the same.

All CLDC HotSpot Implementation command-line options and compilation flags (not just the runtime flags) are defined in `src/vm/share/utilities/Globals.hpp`. There are special factoring macros in that file to allow the appropriate runtime options depending on whether the variable `PRODUCT` is set to *true*. The full set of compilation flags are only available in the “development” build modes, which are the *debug* and *release* build modes listed in the previous section.

Note – Default values for the runtime options are given in `src/vm/share/utilities/BuildFlags.hpp`

Refer to [Section 4.6, “Categories of CLDC HotSpot Implementation Runtime Flags” on page 4-14](#) for a complete breakdown of the command-line options and compilation flags.

4.5 Running CLDC HotSpot Implementation

Once you have built a debug or product version of CLDC HotSpot Implementation, you can invoke it from the command line to run a class compiled from the Java programming language. The path to the executable depends on which platform (operating system and processor) you have built for. The general syntax is as follows, where `<platform>` is one of:

- `win32_i386`
- `win32_i386_ide`
- `linux_i386`
- `linux_arm`
- `ads_arm`
- `ads_bigarm`
- `ads_jazelle`
- `linux_thumb`

For a complete list of supported platforms, see the directories under `%JVMWorkspace%\build\`.

To run CLDC HotSpot Implementation with an executable built in *debug* mode:
`%JVMBuildSpace%\<platform>\bin\cldc_hi_g -classpath C:<location of compiled Java applications>\classes <classname>`

To run CLDC HotSpot Implementation with an executable built in *release* mode:
`%JVMBuildSpace%\<platform>\bin\cldc_hi_r -classpath C:<location of compiled Java applications>\classes <classname>`

To run CLDC HotSpot Implementation with an executable built in *product* mode:
`%JVMBuildSpace%\<platform>\bin\cldc_hi -classpath C:<location of compiled Java applications>\classes <classname>`

If you have built CLDC HotSpot Implementation with ROMizing turned off (that is, with the `ROMIZING=false` option), you need to specify a classpath when running: `%JVMBuildSpace%\<platform>\bin\cldc_hi -classpath %JVMBuildSpace%\classes;C:<location of compiled Java applications>\classes <classname>`

There are numerous command options available, as detailed in the next section, and in [Section 4.6, “Categories of CLDC HotSpot Implementation Runtime Flags”](#) on page 4-14.

4.5.1 Runtime Command Line Options

The syntax for running CLDC HotSpot Implementation is as follows:

```
cldc_hi[_g,_r] [options] class [args...]
```

where `cldc_hi_g` is the command to run the executable built in *debug* mode, `cldc_hi_r` is the command to run the executable built in *release* mode, and `cldc_hi` is the command to run the executable built in *product* mode, and where *options* include:

TABLE 4-4 cldc_hi Command-line Options

<code>-cp -classpath <directory></code>	Set search path for application classes and resources
<code>-comp</code>	Compile all methods
<code>-int</code>	Force interpreted mode
<code>-verbose</code>	Enable verbose output
<code>-? -help</code>	Print this help message
<code>-flags</code>	List all available Runtime Flags
<code>-buildopts</code>	List all build-time options
<code>-definitions</code>	List values of virtual machine symbolic definitions
<code>-errorcodes</code>	List all error codes
<code>-romize</code>	Generate rom image of system classes
<code>-generate</code>	Generate interpreter source file
<code>-generateoptimized</code>	Generate optimized interpreter source file

TABLE 4-4 cldc_hi Command-line Options

<code>-romconfig <file></code>	Name of ROMizer configuration file
<code>+/-GlobalFlag</code>	Turn on/off a boolean Global Flag
<code>=Globalflag NN[K M]</code>	Set an integer Global Flag to value <i>NN</i> . <i>K</i> means Kilobytes, <i>M</i> means Megabytes. For example, <code>=HeapCapacity10M</code> means set HeapCapacity to 10 MB

Note – If the `UseVerifier` command option is used (see TABLE 4-5), you must run the `preverify` tool before running the CLDC HotSpot Implementation executable. The `UseVerifier` option is true by default. The `preverify` tool is installed at `%JVMWorkspace%\build\share\bin\platform name`.

4.5.2 Runtime Command Line Examples

Here is an example of starting up the CLDC HotSpot Implementation virtual machine built in *Product* mode on Windows 2000 to run a classfile implementing “HelloWorld:”

```
%JVMBuildSpace%\win32_i386\dist\bin\cldc_hi.exe -classpath  
%JVMBuildSpace%\classes.zip;C:\tests HelloWorld
```

For example, if you set `JVMBuildSpace=C:\mycldchi_output`, the command would be

```
C:\mycldchi_output\win32_i386\dist\bin\cldc_hi.exe  
-classpath C:\mycldchi_output\classes.zip;C:\tests HelloWorld
```

Other examples:

```
cldc_hi_g +UseCodeFlushing      (Turn code flushing on)  
cldc_hi_g -UseProfiler          (Turn profiler off)  
cldc_hi_g =CompiledCodeFactor15 (Set compiled code factor to 15)
```

4.6 Categories of CLDC HotSpot Implementation Runtime Flags

As explained in [Section 4.4, “CLDC HotSpot Implementation Runtime Flags” on page 4-10,](#) most of the compilation flags and command line options in the system can be treated uniformly. These CLDC HotSpot Implementation runtime flags can be set statically in `Globals.hpp` or set by at the command line during compiling and building. With a slightly different syntax, the same options can be set or turned on or off from the command line when running the CLDC HotSpot Implementation executable.

Most CLDC HotSpot Implementation runtime flags are available only in the “development” build modes of CLDC HotSpot Implementation (that is, the *debug* or *release* build modes.) These are the “non-product” build modes. Refer to the tables below in [Section 4.6.2, “Development Mode Runtime Flags” on page 4-17](#) for a detailed breakdown of these flags.

The flags detailed in [Section 4.6.1, “Product Mode Runtime Flags” on page 4-14](#) on the next page are always available.

4.6.1 Product Mode Runtime Flags

The *product* mode flags in [TABLE 4-5](#) are always available.

TABLE 4-5 Product Mode Command Options

Option	Description	Default
Runtime System		
Configuration options:		
<code>UseVerifier</code>	Should class file verification be performed?	<code>true</code>
<code>VerifyOnly</code>	Verify all jar/zip files in the classpath, then exit.	<code>false</code>
<code>MixedMode</code>	Execution mode: (no effect if! <code>UseCompiler</code>) <code>true</code> -> dynamic recompilation <code>false</code> -> compile methods before executed	<code>true</code>
<code>SlaveMode</code>	Run the virtual machine in slave mode.	<code>false</code>

TABLE 4-5 Product Mode Command Options

Option	Description	Default
CollectBeforeCompilerTest	Do a full GC before compiling each method with <code>-testcompiler</code> .	false
CachedAsyncDataSize	Precreated buffer size for <code>SNI_AllocateReentryData</code> .	8
InterpretationLogSize	How many elements of <code>interpretation_log</code> to examine during timer tick. Set to 0 to disable interpretation log.	INTERP_LOG_SIZE
EnableLookupTableSizeHeuristic	Enable lookup of table size heuristic.	true
CacheJarFileHandles	Enable caching of JAR file handles.	true
EnablePerTaskDebugging	Allow each Isolate to connect to its own debugger.	false
Deterministic	Make virtual machine behave more deterministically.	false
Printing options		
LogVMOutput	Log virtual machine output on file <code>cldc_hi.log</code> .	false
Debugging options:		
VerifyOnly	Verify all jar/zip files in the classpath, then exit.	false
DefaultDebuggerPort	Default port that debugger listens on.	2800
EnablePerTaskDebugging	Allow each Isolate to connect to its own debugger.	false
RemoteTracePort	Enable remote tracing of certain virtual machine parameters by setting port.	-1
Threading		
Configuration options		
StackSize	Minimum stack size in bytes.	4 * K
StackPadding	Total amount of stack padding in bytes.	1 * K+256
StackSizeIncrement	Increment in bytes for stack resizing.	2 * K
StackSizeMaximum	Maximum stack size in bytes.	128 * K
PrintCompilationAtExit	At virtual machine exit, prints a line for each compiled method.	false
StarvationTime	Max time in clock ticks to starve a thread.	200
Compiler		

TABLE 4-5 Product Mode Command Options

Option	Description	Default
Configuration options:		
UseCompiler	Should the compiler be used? (<i>false</i> means interpreted mode.)	true
CompilerAreaPercentage	Maximum percentage of heap to use by JIT compiler.	20
MinimumCompilerAreaPercentage	Minimum percentage of heap to use by JIT compiler.	0
MaxCompilationTime	Suspend compilation if a method takes more than this time to compile (in milliseconds.) MaxCompilationTime can be by re-implementing <code>Os::check_compiler_timer()</code> .	30
MaxMethodToCompile	Don't compile method with more than this amount of bytecodes (in bytes)	6000
Debugging options:		
UseOSR	Should on-stack-replacement be used?	true
GenerousOSR	Generate an extra OSR for the bci that caused compilation?	true
Code optimization:		
OptimizeCompiledCode	Optimize the code generated by the dynamic compiler	true
Object Heap		
Configuration options:		
HeapCapacity	Capacity of object heap in bytes.	1 * M
HeapMin	Initial object heap capacity in bytes. 0 indicates that HeapCapacity will be used.	0
RecommendedFreeHeapPercentage	Recommended percentage of heap to keep free	10
Isolates		
Configuration options:		
TaskFairScheduling	Give each task a fair chance to run regardless of thread priority.	true
SchedulerDivisor	Divide ticks/sec by this to get number of tokens for normal priority task.	2

TABLE 4-5 Product Mode Command Options

Option	Description	Default
ReservedMemory	Reserved memory for the first isolate.	0
TotalMemory	Total memory for the first isolate.	max_jint
Profiler		
Configuration options:		
UseProfiler	Use execution time profiler (see -profile).	false
UseExactProfiler	Use exact(WTK-style) profiler.	false
SaveSerialProfiles	Save a serial of WTK profiles in graph0.prf, graph1.prf, ..., when the virtual machine is restarted inside the same process.	false

4.6.2 Development Mode Runtime Flags

This section details all the *develop mode* flags: flags available in the *Debug* or *Release* build modes. The flags are categorized into five sections, each of which has a table below.

- Generate Options
- Runtime System
- Interpreter
- Compiler
- Object Heap

Each table might be further divided into these subsections:

- Configuration options
- Debugging options
- Printing options

TABLE 4-6 Develop Mode Generate Options

Option	Description	Default
Configuration options		
EnableAllROMOptimizations	Turn on all ROM optimizations.	false
GenerateROMStructs	Generate ROMStructs.h for accessing Romized classes in C code.	false

TABLE 4-6 Develop Mode Generate Options

Option	Description	Default
GenerateFastMemRoutines	Generate fast version of memory handling routines.	true
GenerateInlineAsm	Generate assembly code output as C with inline assembler.	false
MakeROMDebuggable	Turn off ROM optimizations that change bytecode offsets.	false

TABLE 4-7 Develop Mode Runtime System Options

Option	Description	Default
Configuration options		
ThreadSize	Size of instances of <code>com.sun.cldchi.jvm.Thread</code> (used while bootstrapping).	100
GenerateBruteForceICacheFlush	Generate brute-force icache flushing (large number of no-ops).	false
BruteForceICacheFlushSize	Number of bytes in the brute-force icache flushing function.	32*K
UseJarCache	Use JarFile cache, when built with <code>ENABLE_JAR_ENTRY_CACHE=true</code> .	
MakeRestrictedPackagesFinal	If possible, make all classes in a restricted package 'final.'	true
AbortOnInfiniteWait	Abort the virtual machine if all threads are waiting forever.	false
RunFinalizationAtExit	Run all finalizers when virtual machine terminates.	true
SupportFinalization	Support JLS finalization.	false
GenerateCompilerComments	Generate comments from the compiler into relocation information.	false
UseROM	Should ROM compiled stuff be used?	USE_ROM_DEFAULT
GenerateRelaunchableROM	Generate a ROM image that can be relaunched in the same process.	true
EnableROMCompilation	Generate a ROM image that can be relaunched in the same process.	true
RemoveUnusedSymbols	Remove unused symbols, reduce footprint.	true

TABLE 4-7 Develop Mode Runtime System Options

Option	Description	Default
ROMHashTableDepth	Average depth of each bucket in the ROM Symbol/String tables.	8
Verbose	Print additional debugging information.	false
VerbosePointers	Print value of pointers in debugging information (ignored if Verbose=false).	true
Debugging options		
UseProfiler	Use execution time profiler (see -profile).	false
LoadROMDebugSymbols	Java stack traces printed with more information.	true
TraceFakeClasses	Trace creation of fake classes during signature parsing.	false
TraceExceptions	Trace exception throwing (with stack traces).	false
TraceVerifierVerbose	Trace class file verification in verbose mode.	false
GenerateROMComments	Generate comments in the ROM image.	false
VerboseROMComments	More verbose comments in the ROM image.	false
LogROMPercentages	Print percentages in ROMLog.txt file.	true
Printing options		
PrintVerifiedClasses	Print verified methods and classes.	false
PrintMallocFree	Trace calls to C malloc and free.	false
PrintLongFrames	Print locals and expression stack when printing frames.	false
PrintExtraLongFrames	Print very detailed information when printing frames.	false

TABLE 4-8 Develop Mode Interpreter Options

Option	Description	Default
Generation options		
GenerateAssemblyCode	Generate assembly code output.	false
GenerateGNUCode	Generate assembly code output.	false
GenerateOopMaps	Generate assembly code output.	false
AddExternCUnderscore	Add a _ prefix to extern C symbols in assembly code output.	false

TABLE 4-8 Develop Mode Interpreter Options

Option	Description	Default
EnableAlignmentInInterpreter	Align template entries if <code>align_code_base</code> is set.	false
GenerateBruteForceICacheFlush	Generate brute-force icache flushing (large number of no-ops).	false
BruteForceICacheFlushSize	Number of bytes in the brute-force icache flushing function.	32*K
GenerateDebugAssembly	Generate assembly code containing debug assertions.	false
GenerateThumbInterwork	Enable ARM/THUMB interworking in the generated assembly code.	false
Configuration options		
RewritePairs	Rewrite frequent byte code pairs.	false
BytecodeNumber	Bytecode number for various interpreter generation routines.	0
CountLinearExtensions	Get the number of linear extensions of bytecode <code>BytecodeNumber</code> .	false
UseLinearExtension	Use a specific linear extension for bytecode <code>BytecodeNumber</code> .	
BytecodeHistogramCutOff	The bytecode histogram cut-off percent.	1
PairHistogramCutOff	The pair histogram cut-off percent.	1
EnableCPUVariant	Allow interpreter to use CPU variant features if available.	true
Debugging options		
StartTraceAt	Start printing trace at given bytecode number.	0
StopAtBytecode	Stop at given bytecode.	-1
ExplicitNullChecks	Generate explicit null checks in compiled code.	true
UseFastInterpreterMethodEntries	Use specialized interpreter method entries.	true
Printing options		
TraceFastAccessors	Trace use of fast accessors.	false
PrintPairHistogram	Prints a histogram of the executed bytecode pairs.	false
PrintBytecodeHistogram	Prints a histogram of the executed bytecodes.	false

TABLE 4-9 Develop Mode Compiler Options

Option	Description	Default
Configuration options		
CheckStackOverflow	Insert stack overflow check on backward branches.	true
ShareExceptionStubs	Share exception thrower stubs for compiled methods without exception handlers.	true
OptimizeArrayCopy	Use native library for arraycopy.	true
ResolveConstantPoolInCompiler	Try to resolve constant pool entries inside the compiler if possible.	true
LeafMethodStackPadding	The number of bytes in StackPadding reserved for leaf methods.	256
InlineAccessors	Inline accessor methods.	true
UseQuickNativeMethods	Execute quick native methods without creating a Java frame.	true
AbortOnFailedKNILookup	Abort the virtual machine when native code tries to look up non-existent (or renamed) fields using <code>KNI_GetFieldID</code> . This can be used to check if you have accidentally used ROM optimization options to renamed fields that must not be renamed because they are used by <code>KNI_GetFieldID</code> .	false
ExcessiveSuspendCompilation	Always suspend compilation after processing each compilation queue element (for debugging background compilation).	0
Debugging options		
BreakOnOSREntry	Insert breakpoint in OSR entry stubs.	false
TrapALot	Call <code>uncommon_trap</code> frequently.	false
TrapALotInterval	Interval in which the traps are inserted.	1
InstallCompiledCode	Call compiled code after compilation (for debugging).	true
UseTagStoreElimination	Avoid storing tags that are already in memory.	true
VerifyCachedTags	Verify the cached tags in memory.	false
UseStaticFinalImmediates	Use static final immediate information.	true
BreakOnMethodEntry	Insert breakpoint at compiled method entry.	false

TABLE 4-9 Develop Mode Compiler Options

Option	Description	Default
GuardCompileStubs	Insert breakpoint before each CompileStub to prevent accidental fall-through.	false
VerifyFPUStack	Verify x86 FPU register stack empty at start of basic block.	false
VerifyBailoutStack	Verify the stack at bailout time.	false
Printing options		
TimeCompiler	Gather total compilation time.	false
PrintCompilation	Prints a line for each compiled method.	false
PrintCompiledCode	Prints the native code for all compiled methods.	false
PrintCompiledCodeAsYouGo	Prints the native code for compiled methods at time of compilation.	false
OptimizeCompiledCodeVerbose	Print out instructions changed by the code optimizer.	false

TABLE 4-10 Develop Mode Object Heap Options

Option	Description	Default
Configuration options		
YoungGenerationTarget	Target size of young generation (as fraction of heap size). For example, 5 means 20%.	5
YoungGenerationSurvivalTargetPercentage	If the survival rate in a young GC is smaller than this, do not expand the young generation.	10
YoungGenerationAtEndOfHeap	Put young generation at end of the heap.	false
MinimumMarkingStackSize	Minimum number of elements available on marking stack.	2 * K
MinimumCompileSpace	Minimum free heap space to allow compiler invocation.	30 * K
CompilationAbstinenceTicks	Number of ticks to abstain from compilation after incurring compilation or garbage collection.	0
ExcessiveGC	Call collect at every allocate.	false
VerifyGC	Verify the heap before and after garbage collection.	false
Debugging options		

TABLE 4-10 Develop Mode Object Heap Options

Option	Description	Default
CollectALot	Call <code>collect</code> frequently.	false
GCDummies	Dummy objects allocated at bottom of heap ensuring all objects move at garbage collection.	0
Printing options		
PrintAllObjects	Print all objects by iterating over the object heap when virtual machine exits.	false
PrintObjectHistogramData	Print a log for computing the object histogram.	false
PrintLoadedClasses	Print a class after class loading.	false
TraceCodeFlushing	Trace marking/flushing of compiled methods.	false

TABLE 4-11 Develop Mode Isolates Options

Option	Description	Default
Configuration options		
ProhibitCompiledCIB	Prohibits compilation if it must include a specific barrier.	0
EmptyCompiledCIB	Generate marker code only for specific barrier.	0
Debugging options		
StopAtCIBHit	Generate breakpoint at barrier hit in compiled methods.	false
StopAtCompiledCIB	Generate breakpoint at barrier in compiled methods.	0
StopAtCompiledStaticAccess	Generate breakpoint at static var access in compiled methods.	0
StopAtCompiledCIBCclearance	Generate breakpoint at barrier mark clearance in compiled methods.	0
StopAtRealMirrorAccess	Generate breakpoint upon getting the java mirror for the current task.	false
StopAtNullPtrException	Generate compiled code breakpoint upon throwing a null pointer exception.	false

4.6.3 Conditional Mode Runtime Flags

The *conditional* build mode flags in TABLE 4-5 are considered *develop* or *product* depending on the rules in Section 4.2.1, “When Runtime Flags are Available” on page 4-2.

TABLE 4-12 Conditional Mode Generate Options

Option	Description	Default
Configuration options		
GenerateROMImage	Generate the ROMized image to support pre-loading of Java class libraries. The <code>classpath</code> must contain the path to the archived classes that will be ROMized.	false
GenerateSystemROMImage	Generate ROM image of system classes.	false
EnableBaseOptimizations	Do basic optimizations for ROM generator.	true
EnableClassUnloading	Enable removal of unused classes.	false
OptimizeBytecodes	Replace some bytecodes with shorter sequences.	true
PostponeErrorsUntilRuntime	Don't abort romizer on an error caused by invalid class file. Instead, produce a valid ROM image that reports this error at runtime. By default, enabled for application classes.	
LoopPeelingSizeLimit	Do not peel the loop if generated code for first run exceeds this limit (in bytes).	100
OptimizeForwardBranches	Optimize simple forward branches.	USE_OPT_FORWARD_BRANCH
OptimizeLoops	Make the compiler optimize loop code (enables loop peeling).	true
CompiledCodeFactor	Compute the maximum compiled code size using method code size.	15
UseEventLogger	Enable EventLogger, and print event log at virtual machine exit. 0 for no logging, 1 for critical events and 2 for all events.	0
LogEventsToFile	If true, write the event log into event.log. Otherwise dump to tty.	false
RetryCompilation	Retry compilation if CompiledCodeFactor is too small	true

TABLE 4-12 Conditional Mode Generate Options

Option	Description	Default
TestCompiler	Compile all methods specified in the classpath.	false
TestCompileSystemClasses	Also compile the methods in the (romized) system classes in -testcompiler mode.	false
EnableTicks	Enable simulated timer ticks (for scheduling and compilation.	true
TraceNativeCalls	Trace native method calls.	false
TraceCompiledMethodCache	Trace compiled method cache events.	false
TraceDebugger	Trace Java debugger support operations.	false
TraceRomizer	Trace operations by the (source and binary) romizer.	false
TraceThreadEvents	Trace thread events (transfer yield, etc.).	false
TraceThreadsExcessive	Trace more thread events.	false
TraceVerifier	Trace class file verification.	false
TraceStackmaps	Trace stack map generation.	false
TraceStackmapsVerbose	Verbose stack map tracing.	false
TraceVerifierByteCodes	Trace bytecodes during verification.	false
TraceOSR	Trace on-stack-replacements.	false
TraceUncommonTrap	Trace uncommon-traps that are taken.	false
EnableClassUnloading	Enable removal of unused classes.	false
TraceExceptions	Trace exception throwing (with stack traces).	false
Printing options		
TraceClassLoading	Print a line when a class is loaded.	false
TraceClassUnloading	Print a line when a class is unloaded.	false
TraceClassInitialization	Print a line when a class is initialized.	false
TraceGC	Verbose trace of garbage collection.	false
TraceHeapSize	Verbose trace of heap growth, shrinking.	false
TraceCompilerGC	Verbose trace of GC in compiler_area.	false
TraceFailedCompilation	Verbose trace when method fails to compile.	false
TraceFinalization	Verbose trace of finalization behavior.	false
VerboseGC	Print user-level garbage collection information.	false

TABLE 4-12 Conditional Mode Generate Options

Option	Description	Default
TraceJarCache	Verbose trace of JarFile cache.	false
VerboseClassLoading	Print user-level class loading information.	false
TraceBytecodes	Generate interpreter that traces bytecodes.	false
TraceBytecodesCompiler	Generate compiled code that traces bytecodes.	false
TraceBytecodesStart	Start printing trace at bytecode number.	0
TraceBytecodesInterval	Start printing trace at given bytecode number.	1
TraceBytecodesStop	Stop printing trace at bytecode number.	-1
TraceBytecodesVerbose	Stop at given bytecode.	false
TraceMirandaMethods	Print miranda methods (abstract interface methods added silently by the virtual machine.	false
EmbeddedROMHashTables	Embed ROM hashtables in merged constant pool.	true
RewriteROMConstantPool	Rewrite constant pools in ROM to reduce static footprint.	true
MergedConstantPoolLimit	Max size of a merged ConstantPool in ROM.	65535
RemoveDeadMethods	Remove non-app accessible methods in ROM.	false
RenameNonPublicROMClasses	Rename non-public classes in ROM.	false
RenameNonPublicROMSymbols	Rename non-public fields/methods in ROM?	false
AggressiveROMSymbolRenaming	Rename all fields/methods in non-public romized classes.	false
CompactROMFieldTables	Remove redundant entries from field tables of romized classes.	false
CompactROMBytecodes	Replace ROM bytecodes with equivalent but shorter sequences.	true
CompactROMMethodTables	Remove redundant entries from method tables of romized classes.	false
SimpleROMInliner	Inline simple methods to caller, improve speed and footprint.	true
RemoveDuplicatedROMStackmaps	Eliminate redundant entries in romized StackmapList.	true

TABLE 4-12 Conditional Mode Generate Options

Option	Description	Default
<code>RemoveConvertedClassFiles</code>	Remove converted class files from JAR files in classpath.	<code>false</code>
Performance Counters options		
<code>PrintPerformanceCounters</code>	Print a choice group of 'important' perf counters at virtual machine exit.	<code>false</code>
<code>PrintAllPerformanceCounters</code>	Print all performance counters.	<code>false</code>
<code>PrintGCPerformanceCounters</code>	Print performance counters related to GC.	<code>false</code>
<code>PrintLoadingPerformanceCounters</code>	Print performance counters related to class loading and verifier.	<code>false</code>
<code>PrintThreadPerformanceCounters</code>	Print performance counters related to threads and events.	<code>false</code>
<code>PrintIsolateMemoryUsage</code>	Print max memory usage for every isolate	<code>false</code>

4.6.4 Tuning Flags for Performance

To optimize virtual machine performance for a particular implementation, with a unique combination of processor and memory resources, “tuning” of parameters should be undertaken. This involves testing out combinations of settings of related sets of flags, as detailed in Chapters 6 and 8 of the *CLDC HotSpot Implementation Porting Guide*.

Building on Various Platforms

In general, when your environment is set up correctly, you can build CLDC HotSpot Implementation simply by running the command:

```
gnumake
```

The following sections detail any special requirements to build on most platforms that are supported in this release.

5.1 x86 Linux Platform, Batch Mode

On the Linux platform, environment variables such as *JVMWorkSpace* are expressed as `$(JVMWorkSpace)`. Change directory to *JVMWorkSpace/build/linux_i386* and run `gnumake`.

If *JVMBuildSpace* is defined, output is generated under *JVMBuildSpace/build/linux_i386/target/bin*.

If *JVMBuildSpace* isn't defined, output is generated under *JVMWorkSpace/build/linux_i386/target/bin*.

5.2 x86 Win32 Platform, Batch Mode

On the Win32 platform, environment variables such as *JVMWorkSpace* are expressed as `%JVMWorkSpace%`. Change directory to *JVMWorkSpace\build\win32_i386* and run `gnumake`. For example,
`gnumake debug`

If *JVMBuildSpace* is defined, output is generated under *JVMBuildSpace*\build\win32_i386\target\bin

If *JVMBuildSpace* isn't defined, output is generated under *JVMWorkSpace*\build\win32_i386\target\bin.

Even on the Win32 platform, GNU Make assumes that directory paths contain forward slashes / rather than backward slashes \.

Note – It was found that the build process does not work correctly in bash shell under Windows 2000. Instead, use the default command line interpreter CMD.exe (Command Prompt).

5.3 x86 Win32 Platform, Batch Mode with CodeWarrior Compiler

Change directory to *JVMWorkSpace*\build\win32_i386_cw and run gnumake.

If *JVMBuildSpace* is defined, output is generated under *JVMBuildSpace*\build\win32_i386_cw\target\bin.

If *JVMBuildSpace* isn't defined, output is generated under *JVMWorkSpace*\build\win32_i386_cw\target\bin.

5.4 x86 Win32 Platform, IDE Mode

Requirements to build on this platform are as follows.

- javac.exe, java.exe, and jar.exe must be in your *PATH*.
- JDK 1.3 or higher is required. (JDK 1.4 is recommended.)
- ml.exe must be in your *PATH*. (For more info on ml.exe, see [Section 1.5, "Developing in Microsoft Windows OS" on page 1-3](#)).

Create the IDE project file

```
cd build\win32_i386_ide
idetool.bat create
```

The `idetool.bat` batch file will ask a few questions, create the IDE project file, and launch project in Microsoft Visual Studio.

To launch the IDE project later, you can double-click on the icon `build\win32_i386_ide\cldc_hi.dsw`.

To clean the IDE project and all intermediate files, execute this:

```
cd build\win32_i386_ide
idetool.bat clean
```

To see all the available options, run `idetool.bat` with no arguments.

5.5 ARM Linux Platform, Batch Mode

To build on this platform, perform the following step.

1. Change directory to `JVMWorkspace/build/linux_arm` and run `gnumake`.

If `JVMBuildSpace` is defined, output is generated under `JVMBuildSpace/build/linux_arm/target/bin`.

If `JVMBuildSpace` isn't defined, output is generated under `JVMWorkspace/build/linux_arm/target/bin`.

5.6 ADS Linux Platform, Batch Mode

Follow this procedure if the ADS tools are hosted on a Linux workstation:

1. Ensure the ADS compilers (`armcpp`, `tcpp`, and so forth) are in your `PATH`.
2. Set the environment variable `ADS_LINUX_HOST=true`.
3. Change directory to

```
JVMWorkspace/build/ads_arm
and run gnumake.
```

If `JVMBuildSpace` is defined, output is generated under `JVMBuildSpace/build/ads_arm/target/bin`.

If `JVMBuildSpace` isn't defined, output is generated under `JVMWorkspace/build/ads_arm/target/bin`.

5.7 ADS Win32 Platform, Batch Mode

Follow this procedure if the ADS tools are hosted on a Microsoft Windows workstation.

1. Ensure that the ADS compilers (`armcpp`, `tcpp`, and so forth) are in your `PATH`.
2. Change directory to `JVMWorkSpace\build\ads_arm` and run `gnumake`.

If `JVMBuildSpace` is defined, output is generated under `JVMBuildSpace\build\ads_arm\target\bin`.

If `JVMBuildSpace` isn't defined, output is generated under `JVMWorkSpace\build\ads_arm\target\bin`.

5.8 ARM Symbian Platform, Batch Mode

Symbian OS 7.0 SDK is hosted on a Microsoft Windows workstation. To build on this platform, perform the following steps.

1. Obtain Symbian OS 7.0 SDK from Symbian Ltd., and install it in accordance with the accompanying instructions.

The Symbian OS 7.0 SDK instruction describes how to set the environment variable `EPOCROOT`. The CLDC HotSpot Implementation build system requires that you set an additional variable, `EPOCROOT_U`, which should point to the same directory as `EPOCROOT`, but it includes the disk drive name and uses forward slashes.

For example:

```
set EPOCROOT=\Symbian\7.0\  
set EPOCROOT_U=E:/Symbian/7.0
```

2. Install MKS tools and `gnumake` on your Win32 workstation. (Refer to [Section 1.5, "Developing in Microsoft Windows OS"](#) on page 1-3).
3. Set the environment variables `X86_LIB`, `X86_INCLUDE` and `X86_PATH`. (Refer to [Section 1.6, "Environment Variable Summary"](#) on page 1-4).
4. The build system requires the version of `perl.exe` included in the Symbian SDK. Make sure you set your `PATH` variable in the proper order so that version will be used instead of the version of `perl.exe` provided with MKS Toolkit.
5. Change directory to `JVMWorkSpace/build/symbian_arm4` and run `gnumake`.

If `$JVMBuildSpace` is defined, output is generated under
`${JVMBuildSpace}/build/symbian_arm4/target/bin`.

If `$JVMBuildSpace` isn't defined, output is generated under
`${JVMWorkspace}/build/symbian_arm4/target/bin`.

Note – The Microsoft Macro Assembler (`m1.exe`) is no longer required by the Symbian build process. From version 1.1.2 of CLDC HotSpot Implementation, the x86 variants of the Symbian build use an interpreter generated as in-lined assembly code in a C source file, and can be compiled with the standard MSVC++ compiler.

5.9 XScale Platform, Batch Mode

Details of the procedure to build for this platform are given in Section B.1 of the *CLDC HotSpot Implementation Porting Guide*.

5.10 Building the Portable ROMizer

The ROMizer can be built on platforms (such as Solaris) not supported by the virtual machine. For example, follow this step to build the ROMizer on Solaris:

1. Change directory to `JVMWorkspace/build/solaris_c` and run `gnumake`.

If `JVMBuildSpace` is defined, output is generated under
`JVMBuildSpace/build/solaris_c/target/bin`.

If `JVMBuildSpace` isn't defined, output is generated under
`JVMWorkspace/build/solaris_c/target/bin`.

5.11 Building for Other Target Platforms

The procedure is much the same to build for a different platform, such as *MyPlatform*.

1. Create a configuration file for your platform, called `MyPlatform.cfg`, in the directory `JVMWorkspace/build/MyPlatform`.

2. Change to that directory and run gnumake.

5.12 Building and Running Examples

1. Default CMD/batch build on the Win32-X86 platform. Debug build:

```
>cd JVMBuildSpace\build\win32_i386
>gnumake debug
```

2. Clean:

```
> gnumake clean
```

3. Batch build with ROMizing set to false. Debug build:

```
>cd JVMBuildSpace\build\win32_i386
>gnumake ROMIZING=false debug
```

4. Run CLDC HotSpot (built in example 3) from the command line. *Classpath* must be specified.

```
JVMBuildSpace\target\bin\cldc_hi_g.exe
    -classpath %JVMBuildSpace%\classes;C:\classes MyClass
```

Index

A

adaptive compiler, 6
assembler, macro, 2

B

batch mode builds, 2
binary executables, pre-compiled, 1
boolean flags, 10
build modes, 2
BuildFlags.hpp, 3
building and running CLDC Hotspot
Implementation, 1
building in batch mode, 2
building within an IDE, 1
BuildTool.java, 3

C

C++ source and header file directories, 3
C++ source code, 1
CDLC_HI-HowToBuild.html, 1
CLDC Hotspot Implementation command line
options, categories, 14
CLDC Hotspot Implementation, building and
running, 1
CLDC Hotspot Implementation, running, 11
command-line options, 10
compiler and tool requirements, 1
compiler requirements, 1
conditional build flag, 2
Conditional build mode, available options, 24

D

debug build mode, 2
develop build flag, 2
Develop build mode compiler options, 21
Develop build mode generate options, 17, 24
Develop build mode interpreter options, 19
Develop build mode isolates options, 23
Develop build mode object heap options, 22
Develop build mode runtime system options, 18
Develop build mode, available options, 17
directories, distribution, 1
Directory Structure, 1
distribution directories, 1

E

eMbedded Visual C++, Microsoft, 2

G

Globals.hpp, 10
GNU Make, 1

I

interpreter generator, 4

J

jvmconfig.h, 3

M

makefiles, 1, 2

P

- product build flag, 2
- product build mode, 2
- Product build mode, available options, 14

R

- release build mode, 2
- running CLDC Hotspot Implementation, 11
- Runtime System, 5

S

- source code, C++, 1

U

- UseVerifier command option, 13

V

- vm/cpu, details of, 4
- vm/os, details of, 7
- vm/share, details of, 4