# Oracle® Java Micro Edition Embedded Client

Architecture Guide

Release 1.1

**E23813-01**

June 2012

Documentation for all application developers and customizers. This document is the prerequisite to other Oracle Java Micro Edition Embedded Client documentation. It describes fundamental product concepts and facilities.

Oracle Java Micro Edition Embedded Client Architecture Guide, Release 1.1

E23813-01

# Contents

## List of Tables

# Preface

This guide describes concepts that are fundamental to successful use of the Oracle Java Micro Edition Embedded Client.

## Audience

This document is intended for all Oracle Java Micro Edition Embedded Client users. All Oracle Java Micro Edition Embedded Client guides (see "Related Documents") assume their readers are familiar with the concepts and features described in this guide.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

### Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

## Related Documents

For more information, see the following documents in the Oracle Java Micro Edition Embedded Client documentation set:

- *Oracle Java Micro Edition Embedded Client Customization Guide*
- *Oracle Java Micro Edition Embedded Client Developer's Guide*

**Note:** The *Oracle Java Micro Edition Embedded Client Architecture Guide* is a prerequisite for all Oracle Java Micro Edition Embedded Client guides. It defines concepts that are mentioned in the other guides.

## Conventions

The following text conventions are used in this document:

| Convention | Meaning |
|---|---|
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |

# 1 Overview

This chapter describes essential Oracle Java Micro Edition Embedded Client terms and concepts, including the kinds of devices that are the domain of the product, the roles that product users play, host and target platforms, and options for application development.

This chapter includes the following topics:

- Section 1.1, "Domains and Devices"
- Section 1.2, "Users and Documentation"
- Section 1.3, "Components"
- Section 1.4, "Platforms"
- Section 1.5, "Developing Applications"

## 1.1 Domains and Devices

The benefits of writing applications in the Java programming language can be realized across a range of computing domains from corporate applications running on large servers to personal applications hosted by small mobile phones. In all of these environments, a Java virtual machine executes Java bytecodes, which are the basis of Java application portability. However, to exploit the resources of large computers and accommodate the constraints of very small ones, there must be more than one Java platform — a virtual machine and set of libraries that provide services to Java applications.

Oracle Java Micro Edition Embedded Client is one of a family of Java platforms available from Oracle. Other family members include Java Standard Edition (Java SE, for desk- and laptop computers and servers), and Oracle Java Wireless Client, typically used for feature phones. Oracle Java Micro Edition Embedded Client falls between the Standard Edition and the Wireless Client. It is designed for devices whose computational resources are between those of phones and laptop computers. Typical application domains are televisions, Blu-ray Disc players, and set-top boxes; printers, IP phones, ebook readers, and smart utility meters.

Oracle Java Micro Edition Embedded Client and the applications that use it run on a device. A device is usually a specialized computer and peripheral components embedded in a machine or appliance whose users do not think of as a computer. A device is called a *target device* when it is necessary to distinguish it from a *host device* (or host computer), which is a personal computer that is used for developing Oracle Java Micro Edition Embedded Client or Oracle Java Micro Edition Embedded Client applications. A device consists of a CPU, memory, peripheral hardware, and an operating system. The peripheral hardware might include device-specific components in addition to those found on a typical small embedded computer.

## 1.2 Users and Documentation

The Oracle Java Micro Edition Embedded Client documentation set distinguishes the following users:

- Device users: These are the end users of the device that incorporates Oracle Java Micro Edition Embedded Client. They are not aware of Oracle Java Micro Edition Embedded Client's presence. They use the device as a smart meter, a printer, an eBook reader, and so on. Device user documentation is written by the organization that uses Oracle Java Micro Edition Embedded Client to create a device.
- Application developers: Java programmers who build the software that uses Oracle Java Micro Edition Embedded Client and interacts with device users. These developers know about Oracle Java Micro Edition Embedded Client's Java programming interfaces but not its implementation. The primary document for application developers is the Oracle Java Micro Edition Embedded Client *Developer's Guide*.
- Customizer: Customizers adjust the Oracle Java Micro Edition Embedded Client to fit requirements for memory footprint, performance, special hardware, and so on. The primary document for customizers is the Oracle Java Micro Edition Embedded Client *Customization Guide*.

One person might act as multiple users. For example, a customizer might also be an application developer.

## 1.3 Components

Oracle Java Micro Edition Embedded Client includes the components shown in Table 1–1. As noted in the table, some components can be removed to reduce device memory requirements.

***Table 1–1 Headless Configuration Main Components***

| Component | Spec Version | Description |
|---|---|---|
| Connected Device Configuration (CDC) | JSR 218 1.1 | Java virtual machine with interpreter and compiler, plus basic libraries. |
| Foundation Profile (FP) | JSR 219 1.1 | Input/output, utility, security classes that supplement the CDC APIs to the functional level of the Java SE 1.4 platform. |
| Security Optional Package (SecOp) | JSR 219 1.1 | Optional component of Foundation Profile that defines cryptographic APIs. Removable. |
| J2ME Remote Method Invocation (RMI) | JSR 66 1.0 | Remote invocation of Java methods. Removable. |
| Java Data Base Connection for CDC/FP (JDBC) | JSR 169 1.0 | Relational database query and update, subset of full JDBC. Removable. |
| XML API | JSR 280 1.0 | Subsets of XML parsing APIs. Removable. |
| Web Services | JSR 172 1.1 | XML parsing and XML remote procedure calls. Installable. |

## 1.4 Platforms

Besides distinguishing among the Java editions (SE, and so on, see Section 1.1), in the context of Oracle Java Micro Edition Embedded Client, there is a second meaning for "platform". An Oracle Java Micro Edition Embedded Client platform is a computer on which an Oracle Java Micro Edition Embedded Client binary runs. The Oracle Java Micro Edition Embedded Client *Release Notes* describe the platforms supported in this release. There are both host and target platforms. The target platforms for the headless configuration differ from those for the headful configuration, because the latter must include graphics support.

## 1.5 Developing Applications

Oracle Java Micro Edition Embedded Client application development is cross-platform. Target devices run Oracle Java Micro Edition Embedded Client applications, but their limited resources are too limited for application development. Developers code and compile on a host computer. They test by transferring their code to a device or emulator. Debugging and profiling is performed across a network, using the host computer as a console and instrumentation on the device.

Oracle Java Micro Edition Embedded Client applications can be developed with standard command-line tools, such as `javac` and `make` or `ant`. You can also use the NetBeans or Eclipse Integrated Development Environments (IDEs).

# 2 Connected Device Configuration and Foundation Profile

This chapter describes the fundamental components of the Oracle Java Micro Edition Embedded Client, the virtual machine that executes Java applications, two sets of the libraries that applications can use. Together, the virtual machine and core libraries described in this chapter are called the Connected Device Configuration (CDC). Supplementing CDC is the set of libraries called the Foundation Profile.

This chapter includes these topics:

- Section 2.1, "Virtual Machine"
- Section 2.2, "CDC Class Libraries"
- Section 2.3, "Foundation Profile Class Libraries"

## 2.1 Virtual Machine

Java programs are compiled on a host computer into a portable intermediate form called Java bytecodes. In Oracle Java Micro Edition Embedded Client, files containing bytecodes are loaded to the target device where the resident CDC virtual machine (CVM) inspects, decodes, and executes them.

Section 6.2 describes an alternative ahead-of-time compiler.

### 2.1.1 Interpreter

Java compilers generate machine-independent bytecodes instead of machine instructions. The interpreter is like a CPU implemented in software. It decodes and executes bytecodes, independent of what computer they were compiled on.

### 2.1.2 Dynamic Compiler

As the interpreter executes blocks of bytecodes, it tracks the number of times a block is executed. Frequently executed blocks are called hot spots. Periodically during execution, the dynamic compiler creates machine-language versions of hot spots, which are thereafter invoked instead of the interpreter. Compiled code runs about 10 times as fast as interpretation, so the investment in compilation pays off quickly.

For details on how the dynamic compiler works and how you can adjust its operation, see the *Customization Guide*'s Tuning chapter.

### 2.1.3 Memory Management

Automatic reclamation of unusable heap objects (garbage collection) is central to the Java virtual machine. By automating memory reclamation, the garbage collector prevents common and difficult-to-diagnose programmer errors. Failing to release unneeded memory, and prematurely releasing needed memory can stop a system, which is especially serious for an embedded device. The CDC garbage collector is efficient and can be adjusted to minimize interference with user-visible activities.

For details on how the heap works and how you can adjust its configuration, see the *Customization Guide*'s Tuning chapter.

### 2.1.4 Security

Virtual machine security features include:

- Verifying the integrity of class (bytecode) files
- Preventing access to unauthorized data and code
- Preventing stack overflows
- Executing code in a "sandbox" that prevents unauthorized access to system resources.

## 2.2 CDC Class Libraries

The CDC class libraries implement a minimal Java API, including subsets of these Java SE packages:

- `java.lang`: virtual machine system classes, including thread
- `java.util`: Java utilities
- `java.net`: UDP, InetAddress, and URL I/O
- `java.io`: Java file input/output
- `java.text`: minimal support for internationalization
- `java.security`: minimal security and encryption

For CDC class library details, see `http://www.oracle.com/technetwork/java/embedded/documentation/index.html`.

## 2.3 Foundation Profile Class Libraries

The Foundation Profile (FP) supplements the CDC class libraries described in Section 2.2. FP gives Oracle Java Micro Edition Embedded Client application developers approximately the same APIs available in Java SE 1.4.2, minus graphical user interface features.

FP adds classes to CDC's `java.lang`, `java.io`, `java.net`, `java.security`, `java.text`, and `java.util`. It also adds HTTP networking to `javax.microedition.io`.  For FB class library details, see `http://www.oracle.com/technetwork/java/embedded/documentation/index.html`.

# 3 Optional Packages

This chapter describes optional packages that provide useful functions for some, but not all, deployments. Optional packages support remote method invocation, database connectivity, XML processing, encryption, and web services. The *Customization Guide* describes how to remove and install these optional packages to optimize the use of device memory.

This chapter includes the following topics:

## 3.1 J2ME Remote Method Invocation (RMI, JSR 66)

The RMI interface enables distributed applications to invoke each other's methods across a network. The Oracle Java Micro Edition Embedded Client RMI API is a subset of the Java SE RMI API, and includes the following capabilities:

- Full RMI call semantics
- Marshalled object support
- RMI wire protocol
- Export of remote objects
- Client- and server-side distributed garbage collection
- The Activator interface and the client-side activation protocol
- Registry interfaces and export of a Registry remote object

For a complete description, including differences from Java SE RMI, see `http://jcp.org/en/jsr/detail?id=66`

## 3.2 Java Database Connectivity for CDC/FP (JDBC, JSR 169)

JDBC is a Java interface for querying and updating a relational database with SQL statements. JDBC for CDC/FP (which is supplied with Oracle Java Micro Edition Embedded Client) is a subset defined at `http://download.oracle.com/javame/config/cdc/opt-pkgs/api/jsr169/index.html`.

To use JDBC, you must logically connect the interface to an actual database as described in the Oracle Java Micro Edition Embedded Client *Customization Guide*.

## 3.3 XML API (JSR 280)

The XML API supports the XML data format in mobile and embedded devices. It consists of subsets of the following:

- Java API for XML Processing (JAXP)
- Simple API for XML Processing (SAX)
- Streaming API for XML (StAX, JSR 173)
- Document Object Model (DOM) Core
- DOM Events and View

The API definition is at `http://download.oracle.com/javame/config/cldc/opt-pkgs/api/xml/jsr280/index.html`.

## 3.4 Security Optional Package

The security optional package included with Oracle Java Micro Edition Embedded Client consists of `javax.crypto` classes that perform encryption, key generation, key agreement, and Message Authentication Code (MAC) generation. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects.

## 3.5 Web Services (JSR 172)

The Web Services optional package enables applications to communicate with XML-based enterprise web services and to parse XML data.

In Oracle Java Micro Edition Embedded Client, the Web Services package and the XML API package are mutually exclusive. The XML API optional package is installed by default. The *Customization Guide* describes how to swap the packages.

# 4 Customization

This chapter describes how you can customize an Oracle Java Micro Edition Embedded Client binary to fit device and market requirements. You can install and delete components, adjust the security policies, and tune memory consumption and performance. For detailed descriptions refer to the *Customization Guide*.

This chapter includes the following topics:

## 4.1 Removable and Installable Components

If your deployment does not need the following components, you can remove them and save memory:

- RMI (see Section 3.1)
- JDBC (see Section 3.2)
- XML (see Section 3.3)
- Security Optional Package (see Section 3.4)

You can also substitute the Web Services component (see Section 3.5) for the XML component.

## 4.2 Locales and Character Sets

Oracle Java Micro Edition Embedded Client's default definitions of locales and character sets are contained in user-modifiable files. You can add locales and character sets to these files.

## 4.3 Security

When the Security Optional Package is in place (not removed), Oracle Java Micro Edition Embedded Client's security model and is identical to that of Java SE 1.4. The Oracle Java Micro Edition Embedded Client *Customization Guide* has more information on configuring the security features.

## 4.4 Tuning

When you launch the virtual machine, you can specify dozens of options that affect performance. For example, you can set the limit for the size of the heap (which affects the frequency of garbage collection), the amount of memory the compiler can use for compiled code, and so on.

# 5 Application Development

This chapter briefly describes application development in the Java language. Software development with Oracle Java Micro Edition Embedded Client is always cross-platform. You write and build code on a host computer, install and test the result on the target device.

This chapter includes these topics:

## 5.1 Application Model

Oracle Java Micro Edition Embedded Client supports the main and Xlet. The main application model, in which the virtual machine runs a single application. Here is a trivial example:

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

## 5.2 Using the NetBeans Integrated Development Environment (IDE)

You can write, compile, and remotely debug and profile applications with the NetBeans IDE. The *Developer's Guide* has instructions for configuring NetBeans to work with Oracle Java Micro Edition Embedded Client.

## 5.3 Using Command Line Tools

You can use conventional command-line tools provided in the Java Developer's Kit (`javac`, the Java compiler), and chosen by yourself, for example, a programming editor and source code repository.

# 6 Tools

This chapter describes tools provided with the Oracle Java Micro Edition Embedded Client that assist application and system development tasks, for example, launching applications, precompiling selected code, and diagnosing memory leaks.

This chapter includes these topics:

- Section 6.1, "cvm Launcher"
- Section 6.2, "Ahead-of-time (AOT) Compiler"
- Section 6.3, "Memory Inspection"

## 6.1 cvm Launcher

The `cvm` launcher is a command line tool that starts the Oracle Java Micro Edition Embedded Client running an application. Each compiled application is a class file that has a method called `main()`. After initializing itself, the virtual machine invokes the application's `main()`. When `main()` returns, the virtual machine and `cvm` exit.

`cvm` has many options that can affect performance, enable debugging, and so on. They are documented in Appendix A.

### 6.1.1 Launching a Java Application

`cvm`, the CDC application launcher is similar to `java`, the Java SE application launcher. For the Oracle (Java ME) Embedded Client, see the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about using `cvm` to launch Java applications for the Oracle Java ME Embedded Client.

Many of `cvm`'s command-line options are borrowed from `java`. The basic method of launching a Java application is to specify the top-level application class containing the `main()` method on the `cvm` command-line. For example,

```
% cvm HelloWorld
```

By default, `cvm` looks for the top-level application class in the current directory. Alternatively, the `synonymous -cp` and `-classpath` command-line options specify a list of locations where `cvm` searches for application classes instead of the current directory. For example,

```
% cvm -cp /mylib/archive.zip HelloWorld
```

Here `cvm` searches for `HelloWorld` in an archive file */mylib/archive*`/.zip`. See Section 6.1.2 for more information about class search paths.

The `-help` option displays a brief description of the available command-line options. Appendix A provides a complete description of the command-line options available for `cvm`.

## 6.1.2 Class Search Path Basics

The Java runtime environment uses various search paths to locate classes, resources and native objects at runtime. This section describes the two most important search paths: the Java class search path and the native method search path.

### 6.1.2.1 Java Class Search Path

Java applications are collections of Java classes and application resources that are built on one system and then potentially deployed on many different target platforms. Because the file systems on these target platforms can vary greatly from the development system, Java runtime environments use the *Java class search path* as a flexible mechanism for balancing the needs of platform-independence against the realities of different target platforms.

The Java class search path mechanism allows the Java virtual machine to locate and load classes from different locations that are defined at runtime on a target platform. For example, the same application could be organized in one way on a MacOS system and another on a Linux system. Preparing an application's classes for deployment on different target systems is part of the development process. Arranging them for a specific target system i s part of the deployment process.

The Java class search path defines a list of locations that the Java virtual machine uses to find Java classes and application resources. A location can be either a file system directory or a `jar` or Zip archive file. Locations in the Java class search path are delimited by a platform-dependent path separator defined by the `path.separator` system property. The Linux default is the colon ":" character.

The Java SE documentation[1] describes three related Java class search paths:

- The *system* or *bootstrap classes* comprise the Java platform. The *system class search path* is a mechanism for locating these system classes. The default system search path is based on a set of `jar` files located in *JRE*`/lib`.
- The *extension classes* extend the Java platform with optional packages like the JDBC Optional Package. The *extension class search path* is a mechanism for locating these optional packages. `cvm` uses the `-Xbootclasspath` command-line option to statically specify an extension class search path at launch time and the `sun.boot.class.path` system property to dynamically specify an extension class search path. The CDC default extension class search path is *CVM*`/lib`, except for some of the provider implementations for the security optional packages described in Section 3.4 which are stored in *CVM*`/lib/ext`. The Java SE default extension class search path is *JRE*`/lib/ext`.
- The *user classes* are defined and implemented by developers to provide application functionality. The *user class search path* is a mechanism for locating these application classes. Java virtual machine implementations like the CDC Java runtime environment can provide different mechanisms for specifying an Java class search path. `cvm` uses the `-classpath` command-line option to statically specify an Java class search path at launch time and the `java.class.path` system property to dynamically specify an user class search path. The Java SE

[1] See the tools documentation at `http://download.oracle.com/javase/1.4.2/docs/tooldocs/tools.html` for a description of the J2SDK tools and how they use Java class search paths.

application launcher also uses the `CLASSPATH` environment variable, which is *not* supported by the CDC Java runtime environment.

#### 6.1.2.2 Native Method Search Path

The CDC HotSpot Implementation virtual machine uses the Java Native Interface[2] (JNI) as its native method support framework. The JNI specification leaves the platform-level implementation of native methods up to the designers of a Java virtual machine implementation. For the Linux-based CDC Java runtime environment described in this runtime guide, a JNI native method is implemented as a Linux shared library that can be found in the native library search path defined by the `java.library.path` system property.

**Note:** The standard mechanism for specifying the native library search path is the `java.library.path` system property. However, the Linux dynamic linking loader may cause other shared libraries to be loaded implicitly. In this case, the directories in the `LD_LIBRRARY_PATH` environment variable are searched *without* using the `java.library.path` system property. One example of this issue is the location of the Qt shared library. If the target Linux platform has one version of the Qt. shared library in `/usr/lib` and the CDC Java runtime environment uses another version located elsewhere, this directory must be specified in the `LD_LIBRRARY_PATH` environment variable.

Here is a simple example of how to build and use an application with a native method. The mechanism described below is very similar to the Java SE mechanism.

1. Compile a Java application containing a native method.

   ```
   % javac -source 1.4 -target 1.4 -bootclasspath lib/btclasses.zip HelloJNI.java
   ```

2. Generate the JNI stub file for the native method.

   ```
   % javah -bootclasspath lib/btclasses.zip HelloJNI
   ```

3. Compile the native method library.

   ```
   % gcc HelloJNI.c -shared -I${CDC_SRC}/src/share/javavm/export \
       -I${CDC_SRC}/src/linux/javavm/include -o libHelloJNI.so
   ```

   This step requires the CDC-based JNI header files in the CDC source release.

4. Relocate the native method library in the `test` directory.

   ```
   % mkdir test
   % mv libHelloJNI.so test
   ```

5. Launch the application.

   ```
   % cvm -Djava.library.path=test HelloJNI
   ```

   If the native method implementation is not found in the native method search path, the CDC Java runtime environment throws an `UnsatisfiedLinkError`.

[2] See the *Java Native Interface: Programmer's Guide and Specification*.

## 6.2 Ahead-of-time (AOT) Compiler

Especially relevant to the embedded marketplace, Oracle Java Micro Edition Embedded Client provides an AOT compiler that produces non-portable machine language images. These load more quickly than bytecodes and eliminate the overhead of tracking and compiling hot spots at run-time.

AOT compilation must be used carefully because the generated code is not subject to Java run-time security checks. The *Customization Guide* describes AOT compilation in detail.

## 6.3 Memory Inspection

If you suspect a memory leak, you can use Oracle Java Micro Edition Embedded Client tools to diagnose the problem. `jhat` (Java heap analysis tool), displays memory objects in form that can be explored with a web browser. You can use `cvmsh` to send virtual machine diagnostic commands to a running Oracle Java Micro Edition Embedded Client. Refer to the *Developer's Guide* describes for more information about both tools.

# A cvm Reference

This appendix describes the `cvm` command in detail.

This appendix includes these topics:

- Section A.1, "Synopsis"
- Section A.2, "Description"
- Section A.3, "Options"

## A.1 Synopsis

```
cvm [-options] class [options ...]
cvm [-options] -jar jarfile [options ...]
```

## A.2 Description

`cvm` launches a Java application. It does this by starting a Java virtual machine, loading its system classes, loading a specified application class, and then invoking that class's `main` method, which must have the following signature:

```
public static void main(String args[])
```

The first non-option argument to `cvm` is the name of the top-level application class with a fully qualified class name that contains the `main` method. The Java virtual machine searches for the main application class, and other classes used, in three locations: the *system class path*, the *extension class path* and the *user class path*. See the Oracle Java Micro Edition Embedded Client *Customization Guide* for more information about Java class paths. Non-option arguments after the main application class name are passed to the `main` method.

If the `-jar` *jarfile* command-line option is used, `cvm` launches the application in the `jar` file. The manifest of the `jar` file must contain a line of the form `MainClass:`*package.classname*. The *classname* string identifies the class having the `main` method which serves as the application's starting point.

The Oracle Java Micro Edition Embedded Client *Developer's Guide* has more information about launching Java applications with `cvm`.

## A.3 Options

`cvm` borrows some of its command-line options from `java`, the Java SE application launcher. Other options are unique to `cvm` and may require certain build options to enable the necessary run-time features. For command-line options that take a *size* parameter, the default units for size are bytes. Append the letter `k` or `K` to indicate kilobytes, or `m` or `M` to indicate megabytes.

Table A–1 describes the command-line options that are shared with the Java SE application launcher.

***Table A–1 Java SE Command-Line Options***

| **Option** | **Description** |
|---|---|
| `-help` | Display usage information and exit. |
| `-showversion` | Display product version information and continue. |
| `-version` | Display product version information and exit. |
| `-fullversion` | Display build version information and exit. |
| `-D`*`property`*`=`*`value`* | Set a system property value. See Appendix B for a description of security properties for Oracle Java Micro Edition Embedded Client. |
| `-classpath` *`classpath`*<br>`-cp` *`classpath`* | Specify an alternate user class path. The default user class path is the current directory. See Section 6.1.2 for more information on classpaths. |
| `-Xbootclasspath[/a \| /p]:`*classpath* | Specify the extension class path. `/a` appends *classpath* list to the default path. `/p` prepends *classpath* list to the default path. See Section 6.1.2 for more information on classpaths. |
| `-Xms`*size* | Set the start size of the memory allocation pool (heap). This value must be greater than 1000 bytes.<br>The default value is 2M. |
| `-Xmx`*size* | Set the maximum heap size.<br>The default value is 7M. |
| `-Xmn`*size* | Set the minimum heap size.<br>The default value is 1M. |
| `-Xss`*size* | Each Java thread has two stacks: one for Java code and one for native code. The maximum native stack size of the main thread is determined by the native application launcher (for example, shell or operating system.). For subsequent threads, the maximum native stack size is set by the `-Xss` option, although this can be ignored by the underlying operating system. See Table A–4 for a description of the command-line options for controlling the size of the Java stack.<br>The default value is `0` which indicates that the value is actually set by the native environment. |

***Table A–1 (Cont.) Java SE Command-Line Options***

| **Option** | **Description** |
|---|---|
| `-enableassertions [:<`*`package`*`>… \| :<`*`class`*`> ]`<br>`-ea [:<`*`package`*`>... \| :<`*`class`*`>]` | Enable Java assertions. These are disabled by default. With no arguments, this switch enables assertions for all user classes. With one argument ending in ..., the switch enables assertions in the specified package and any subpackages. If the argument is simply ..., the switch enables assertions in the unnamed package in the current working directory. With one argument not ending in ..., the switch enables assertions in the specified class.<br><br>If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. So, for example, to run a program with assertions enabled only in the package `com.wombat.fruitbat` (and any subpackages), the following command could be used:<br><br>`% cvm -ea:com.wombat.fruitbat …` <*MainClass*><br><br>The `-enableassertions` and `-ea` switches apply to all class loaders and to system classes (which do not have a class loader). There is one exception to this rule: in their no-argument form, the switches do not apply to system. This makes it easy to turn on assertions in all classes except for system classes. The `-enablesystemassertions` option enables asserts in all system classes (that is, it sets the default assertion status for system classes to true). To run a program with assertions enabled in the package `com.wombat.fruitbat` but disabled in class `com.wombat.fruitbat.Brickbat`, the following command could be used:<br><br>`% cvm -ea:com.wombat.fruitbat… \`<br>`    -da:com.wombat.fruitbat.Brickbat` <*MainClass*> |
| `-disableassertions [:<`*`package`*`>… \| :<`*`class`*`> ]`<br>`-da [:<`*`package`*`>... \| :<`*`class`*`> ]` | Disable Java assertions. This is the default behavior.<br><br>With no arguments, `-disableassertions` or `-da` disables assertions. With one argument ending in ..., the option disables assertions in the specified package and any subpackages. If the argument is simply ..., the switch disables assertions in the unnamed package in the current working directory. With one argument not ending in ..., the switch disables assertions in the specified class.<br><br>The `-disableassertions` and `-da` switches apply to all class loaders and to system classes that do not have a class loader. There is one exception to this rule: in their no-argument form, the switches do not apply to system. This makes it easy to turn on assertions in all classes except for system classes. A separate switch is provided to enable assertions in all system classes. See the description of the `-disablesystemassertions` option. |
| `-enablesystemassertions`<br>`-esa` | Enable assertions in all system classes (sets the default assertion status for system classes to true). |
| `-disablesystemassertions`<br>`-dsa` | Disable assertions in all system classes. |

Table A–2 describes the Oracle Java Micro Edition Embedded Client-specific command-line options.

***Table A–2 CDC-Specific Command-Line Options***

| Option | Description |
|---|---|
| `-XbuildOptions` | Display build options and exit. |
| `-XshowBuildOptions` | Display build options and continue. |
| `-XappName=`*value* | Specify the application name for QPE. This is used to identify the `cvm` process for native application management and control. |
| `-Xverify:[all \| remote \| none]` | Perform class verification.<br>■ `all` verify all classes.<br>■ `remote` verify all but preloaded and system classes.<br>■ `none` do not perform class verification.<br>The default value is `remote`. If `-Xverify` is used without any arguments, the value is `all`. |
| `-XfullShutdown` | Make sure all resources are freed and the VM destroyed upon exit. This is the default for non-process-model operating systems, but is not needed for process-model operating systems, such as Linux. |
| `-Xgc:`*suboption* | Specify GC-specific options. The default GC is the generational garbage collector described in the Oracle Java Micro Edition Embedded Client *Customization Guide*. See Table A–3 for a description of the suboptions.<br>Other garbage collectors are unsupported. |
| `-Xopt:`*suboption* | Control the Java stack. See Table A–4 for a description of the suboptions. The different suboptions can be appended into a single argument with name/value pair separated by commas. |
| `-XtimeStamping` | Enable timestamping. |
| `-Xtrace:`*flags* | Turn on trace flags. Table A–5 shows the hexadecimal values to turn on each trace flag. To turn on multiple flags, bitwise-OR the values of all the flags you want to turn on, and use that result as the `-Xtrace` value. Requires the `CVM_TRACE=true` build option. (Not enabled in binary edition.) |
| `-Xjnicheck` | `-Xjnicheck` performs additional checks for Java Native Interface (JNI) functions. Specifically, the virtual machine validates the parameters passed to the JNI function as well as the runtime environment data before processing the JNI request. Any invalid data encountered indicates a problem in the native code, and the Java Virtual Machine will terminate with a fatal error in such cases.<br>This feature is useful when many new native entry points have been added. It can help avoid crashes from mismatched parameters to native routines.<br>Expect a performance degradation when this option is used. |

Table A–3 describes the suboptions for the `-Xgc` command-line option.

***Table A–3 -Xgc: Suboptions***

| Option | Description |
|---|---|
| `maxStackMapsMemorySize=`*size* | Set the size of the stack map cache. The default value is `0xFFFFFFFF`. |
| `stat` | Collect and display garbage collection statistics. |
| `youngGen=`*size* | Set the size of the young object generation.<br>NOTE: this option is specific to the default generational collector.<br>The default value is `1M`. |

Table A–4 describes the suboptions for the `-Xopt` command-line option.

***Table A–4 -Xopt: Suboptions***

| Suboption | Description |
|---|---|
| `stackMinSize=`*size* | Set the initial size of the Java stack, from <32…65536>.<br>The default for JIT-based systems is `3K` and the default for non-JIT based systems is `1K`. |
| `stackMaxSize=`*size* | Set the maximum size of the Java stack, from `<1024…1048576>`. The default is `128K`. |
| `stackChunkSize=`*size* | Set the amount the Java stack grows when it must expand `<32…65536>`. The default for JIT-based systems is `2K` and the default for non-JIT based systems is `1K`. |
| `forceDoubleRounding` | Applicable only to x86 platforms. If true (the default) double-precision floating point results are rounded. |
| `useStrictFP` | Applicable only to x86 platforms. Enable or disable Java SE `strictfp` semantics, which improves portability of applications that use floating point, possibly reducing precision on some platforms. The default is true. |
| `useSSE` | Applicable only to x86 platforms. Enable Streaming SIMD Extensions, which improves the speed of floating point add, sub, mul, div and a few other operations. The default is '2' which means SSE extension version 2. The value '0' disables the extensions. All other values map to '2'. |

Table A–5 describes the flags used by the `-Xtrace` command-line option. This option is useful for run-time development purposes only and is not enabled in the binary edition.

***Table A–5 -Xtrace: Flags (Not Enabled in Binary Edition)***

| Value | Description |
|---|---|
| `0x00000001` | Opcode execution. |
| `0x00000002` | Method execution. |
| `0x00000004` | Internal state of the interpreter loop on method calls and returns. |
| `0x00000008` | Fast common-case path of Java synchronization. |
| `0x00000010` | Slow rare-case path of Java synchronization. |
| `0x00000020` | Mutex locking and unlocking operations. |
| `0x00000040` | Consistent state transitions. Garbage Collection (GC)-safety state only. |
| `0x00000080` | GC start and stop notifications. |
| `0x00000100` | GC root scans. |
| `0x00000200` | GC heap object scans. |
| `0x00000400` | GC object allocation. |
| `0x00000800` | GC algorithm internals. |
| `0x00001000` | Transitions between GC-safe and GC-unsafe states. |
| `0x00002000` | Class static initializers. |
| `0x00004000` | Java exception handling. |

***Table A–5 (Cont.) -Xtrace: Flags (Not Enabled in Binary Edition)***

| Value | Description |
|---|---|
| `0x00008000` | Heap initialization and destruction, global state initialization, and the safe exit feature. |
| `0x00010000` | Read and write barriers for GC. |
| `0x00020000` | Generation of GC maps for Java stacks. |
| `0x00040000` | Class loading. |
| `0x00080000` | Class lookup in VM-internal tables. |
| `0x00100000` | Type system operations. |
| `0x00200000` | Java code verifier operations. |
| `0x00400000` | Weak reference handling. |
| `0x00800000` | Class unloading. |
| `0x01000000` | Class linking. |

Table A–6 describes the command-line options available with the `CVM_JVMTI` build option. See the Oracle Java Micro Edition Embedded Client *Developer's Guide* for an example of how to use these command-line options.

***Table A–6 JVMTI Options***

| Option | Description |
|---|---|
| `-Xdebug` | Enable VM-level debugging support. |
| `-Xrun`*lib*`:[help]` \| `[`*option*`=`*value*`, ...]` | Enable feature in shared library. For example, `hprof` profiling support. |

Table A–7 describes the command-line options available with the `CVM_JIT=true` build option. The Oracle Java Micro Edition Embedded Client *Customization Guide* describes these options and the concepts behind them.

***Table A–7 -Xjit: Options***

| Option | Default | Description |
|---|---|---|
| `bcost=`*cost* | 4 | Cost of a backward branch, between `<0...32767>`. |
| `climit=`*cost* | 20000 | The popularity threshold for a given method, between `<0...65535>`. The VM compares a per-method count based on `bcost`, `icost` and `mcost` against this threshold to determine when to compile a given method. |
| `codeCacheSize=`*value* | 512k | Size of code cache where compiled methods are stored, between `<0...32M>`. |
| `compile=`*suboption* | policy | When to compile methods. See Table A–9 for descriptions of the suboptions for `compile`. The default policy is based on the suboption defaults listed in this table. |
| `icost=`*cost* | 20 | Cost of an interpreted-to-interpreted method call, between `<0...32767>`. |
| `inline=`*suboption* | all | Perform method inlining when compiling. See Table A–8 for descriptions of the suboptions for `inline`. |

***Table A–7 (Cont.) -Xjit: Options***

| Option | Default | Description |
|---|---|---|
| `lowerCodeCacheThreshold=`*percentage* | 90% | Lower code cache threshold, between `<0%..100%>`. The dynamic compiler decompiles methods until the code cache reaches this threshold. |
| `maxCompiledMethodSize=`*value* | 65535 | Maximum size of a compiled method, between `<0...64K>`. |
| `maxInliningCodeLength=`*value* | 68 | Maximum size of an inlined method, between `<0...1000>`. This value is used as a threshold that proportionally decreases with the depth of inlining. Therefore, shorter methods are inlined at deeper depths. In addition, if the inlined method is less than *value*`/2`, the dynamic compiler allows unquickened opcodes in the inlined method. |
| `maxInliningDepth=`*value* | 12 | Maximum inlining depth of inlined methods/frames, between `<0...1000>`. |
| `maxWorkingMemorySize=`*value* | 512k | Maximum working memory size for the dynamic compiler, between `<0...64M>`. |
| `mcost=`*cost* | 50 | Cost for transitioning between a compiled method and an interpreted method, and vice versa. Between `<0..32767>`. |
| `minInliningCodeLength=`*value* | 16 | The floor value for `maxInliningCodeLength` when its size is proportionally decreased at greater inlining depths. |
| `policyTriggeredDecompilations=`*boolean* | true | Policy triggered decompilations. If `false`, then never decompiles a method to make room for more compilations. Methods remain compiled until the class is unloaded, even if the code cache is full. |
| `trace=`*suboption* | | Set dynamic compiler trace options. See Table A–10. |
| `upperCodeCacheThreshold=`*percentage* | 95 | Upper code cache threshold, between `<0%...100%>`. The dynamic compiler starts decompiling methods during a GC when the code cache passes this threshold unless `policyTriggeredDecompilations=false`. |
| `XregisterPhis=`*boolean* | true | *Unsupported.* |
| `XcompilingCausesClassLoading=`*boolean* | false | *Unsupported.* |
| `Xpmi=`*boolean* | true | *Unsupported.* |
| `XregisterLocals=`*boolean* | true | *Unsupported.* |
| `aot=`*boolean* | `true` | Enable/disable AOTC. |
| `aotFile=`*file* | | AOTC file path. |
| `recompileAOT=`*boolean* | `false` | Recompile AOTC code when this option is set to `true`. The existing AOTC code is replaced when this option is used. |
| `aotCodeCacheSize=`*size* | 672K | Size for the code cache used for AOTC. |
| `aotMethodList=`*file* | | File containing a list of methods to be compiled and saved for AOTC. |

Table A–8 describes the command-line options for selecting when to inline methods.

***Table A–8 -Xjit:inline= Suboptions***

| Suboption | Description |
|---|---|
| `all` | Enable all the options listed below to perform inlining whenever possible. The default. |
| `none` | Do not perform inlining. |
| `virtual` | Perform inlining on virtual methods. |
| `nonvirtual` | Perform inlining on nonvirtual methods. |
| `vhints` | Virtual hints. Use hints gathered while interpreting a method to choose a target method to get inlined when an `invokevirtual` opcode is compiled. |
| `ihints` | Interface hints. Use hints gathered while interpreting a method to choose a target method for inlining when an `invokeinterface` opcode is compiled. |
| `Xvsync` | Inline virtual synchronized methods. Off by default. *Unsupported.* |
| `Xnvsync` | Inline non-virtual synchronized methods. Off by default. *Unsupported.* |
| `Xdopriv` | Inline privileged methods specified by `java.security.AccessController.doPrivileged()`. On by default. *Unsupported.* |

Table A–9 describes the top-level command-line options that control dynamic compiler policies.

***Table A–9 -Xjit:compile= Suboptions***

| Suboption | Description |
|---|---|
| `policy` | Compile according to existing compilation policy parameters such as `icost` and `climit`. The default. |
| `all` | Compile all methods aggressively. *Note:* this hurts performance and should be used only for testing the dynamic compiler. |
| `none` | Do not compile any methods. |

Table A–10 describes the command-line options for controlling dynamic compiler tracing. These options require a build with `CVM_TRACE_JIT=true`. These options are experimental and unsupported.

***Table A–10 -Xjit:trace= Options***

| Suboption | Description |
|---|---|
| `bctoir` | Print information regarding the conversion of Java bytecodes to the JIT internal representation (IR), including a complete dump of all IR nodes. |
| `codegen` | Print the generated code in a format similar to the assembler language of the target processor. If the build option `CVM_JIT_DEBUG=true`, then this also prints the `JavaCodeSelect` rule used to generate the code interspersed with the generated code. |
| `inlining` | Print method inlining information during the bytecode to IR pass, such as which methods were inlined and which ones were not. |
| `iropt` | Print information about optimizations done in the bytecode to IR pass. |
| `osr` | Print a message when compilation of a method is triggered by on stack replacement (OSR). |
| `stats` | Print statistics gathered during compilation. |
| `status` | Print a line of status each time a method is compiled. The output includes the name of the method and whether it was compiled successfully. |

# B System Properties

This appendix describes system properties supported by the Oracle Java Micro Edition Embedded Client.

In addition to the standard base Java SE system properties (see Table B–2), Oracle Java Micro Edition Embedded Client supports the additional system properties described in Table B–1.

***Table B–1 Oracle Java Micro Edition Embedded Client System Properties***

| System Property | Default Value | Description |
|---|---|---|
| `cdcams.decorations` | `false` | Display native window decorations. |
| `cdcams.presentation` | No default | Top-level presentation mode class. |
| `cdcams.repository` | *CVMHOME*`/repository` | Location of application repository. |
| `cdcams.verbose` | `false` | Display extra diagnostic information. |
| `com.sun.midp.implementation` | No default | For dual-stack builds only, a space-separated list of jar files that provide a MIDP implementation. |
| `com.sun.package.spec.version` | "1.4.2" | Indicates the Java SE equivalent version for core class interfaces. Example: `"1.4.2"` |
| `file.encoding` | UTF-8 | Character encoding for the default locale. |
| `file.encoding.pkg` | `sun.io` | Package that contains the converters that handle converting between local encodings and Unicode. |
| `java.ext.dirs` | *CVMHOME*`/lib` | Specifies one or more directories to search for installed optional packages, each separated by `File.pathSeparatorChar`. |
| `java.vm.info` | `"mixed mode"` | Indicates if virtual machine was built with just-in-time-compiler support.<br>Values: `"mixed mode"` or `"interpreter loop"` |
| `microedition.commports` | No default | Comma-delimited list of available communications ports |

**Table B–1 (Cont.) Oracle Java Micro Edition Embedded Client System Properties**

| System Property | Default Value | Description |
|---|---|---|
| `microedition.configuration` | `cdc` | Java ME configuration |
| `microedition.encoding` | `ISO_LATIN_1` | Unicode character encoding |
| `microedition.hostname` | No default | Host platform |
| `microedition.locale` | `en-US` | System locale |
| `microedition.platform` | `j2me` | Java platform |
| `microedition.profiles` | No default | Java ME profile |
| `microedition.securerandom.nofallback` | `false` | Disable the mechanism that allows the CDC Java run-time environment to fallback to using `/dev/urandom` if `/dev/random` does not have enough entropy to work properly. See the Oracle Java Micro Edition Embedded Client *Customization Guide* for more information. |
| `sun.arch.data.model` | Platform-specific | Platform word size.<br>Examples: `"32"`, `"64"`, `"unknown"` |
| `sun.boot.class.path` | No default | Default boot class path. In particular, includes jar files from the `lib` directory. It includes jar files for removable modules. Read-only property. |
| `sun.boot.library.path` | Specified on command line | From here, the VM loads VM libraries (like those related to JVMTI) and any libraries needed for classes on the `-bootclasspath`. Read-only property. |
| `sun.cpu.endian` | Platform-specific | Endianess of CPU, `"little"` or `"big"`. |
| `sun.io.unicode.encoding` | Platform-specific, follows `sun.cpu.endian` | For example `"unicodeLittle"`. |
| `sun.java2d.fontpath` | No default | User-defined path to fonts. Prefix the path value with `"append:"` or `"prepend:"` to specify if it should be searched before or after the JRE-defined font directories. Read-only property. |
| `sun.misc.product` | "Oracle Java Micro Edition Embedded Client" | `"Oracle Java Micro Edition Embedded Client"` |
| `user.country` | US | Country (system dependent). |
| `user.language` | en | Two-letter language code of the default locale (system dependent). |

*Table B–1 (Cont.) Oracle Java Micro Edition Embedded Client System Properties*

| System Property | Default Value | Description |
|---|---|---|
| `user.region` | US | Two-letter country code of the default locale (system dependent). |
| `user.timezone` | UTC | Time zone (system dependent) |
| `user.variant` | No default | Variant (more specific than country and language). |

Table B–2 lists the base Java SE properties defined by the CDC specification and supported by Oracle Java Micro Edition Embedded Client. These properties have no default values.

*Table B–2 Base Java SE System Properties in Oracle Java Micro Edition Embedded Client*

| System Property | Description |
|---|---|
| `file.separator` | File separator ("/" on UNIX) |
| `java.class.path` | Java class path |
| `java.class.version` | Java class format version number |
| `java.compiler` | Name of JIT compiler to use |
| `java.ext.dirs` | Path of extension directory or directories |
| `java.home` | Java installation directory |
| `java.io.tmpdir` | Default temp file path |
| `java.library.path` | Path from which to load native libraries. Default is absolute path to `lib` directory. |
| `java.specification.name` | Java Runtime Environment specification name |
| `java.specification.vendor` | Java Runtime Environment specification vendor |
| `java.specification.version` | Java Virtual Machine specification version |
| java.vendor | Java Runtime Environment vendor |
| `java.vendor.url` | Java vendor URL |
| `java.version` | Java Runtime Environment version |
| `java.vm.name` | Java Virtual Machine implementation name |
| `java.vm.specification.name` | Java Virtual Machine specification name |
| `java.vm.specification.vendor` | Java Virtual Machine specification vendor |
| `java.vm.vendor` | Java Virtual Machine implementation vendor |
| `java.vm.version` | Java Virtual Machine implementation version |
| `line.separator` | Line separator ("\n" on UNIX) |
| `os.arch` | Operating system architecture |
| `os.name` | Operating system name |
| `os.version` | Operating system version |
| `path.separator` | Path separator (":" on UNIX) |
| `user.dir` | User's current working directory |
| `user.home` | User's home directory |

*Table B–2 (Cont.) Base Java SE System Properties in Oracle Java Micro Edition Embedded Client*

| System Property | Description |
|---|---|
| `user.name` | User's account name |

# C

# Serial Port Configuration Notes

This appendix describes how to configure an OS-level serial port on a Linux device so that a Java application can use the `javax.microedition.io.CommConnection` interface to access the corresponding logical serial port connection.

This appendix includes these topics:

- Section C.1, "Serial Port Setup"
- Section C.2, "OS-Level Testing"

---

**Note:** While this example is based on the RS-232 serial interface implementation of `CommConnection` in `com.sun.cdc.io.j2me.comm.Protocol`, an alternate implementation could use the `CommConnection` interface to support other forms of serial communication such as IrDA.

---

***Table C–1 Serial Communications References***

| Interface | Document |
|---|---|
| RS-232 serial communications | `http://tldp.org/HOWTO/Serial-HOWTO-4.html` |
| `minicom` serial communications program | `minicom` |
| Serial port configuration | `setserialport` |
| Serial port driver interface | `ttyS` |

## C.1 Serial Port Setup

1. Setup a serial cable connection between two Linux computers.
2. Become super user, then configure the serial port to use IRQ 4.

   ```
   # setserial /dev/ttyS0 irq 4
   ```

3. Change the file access permissions for the serial port and the lock file.

   ```
   # chmod 777 /dev/ttyS0 /var/lock
   ```

   This allows other users to access the serial port.

4. Launch the `minicom` serial communications program in setup mode.

   ```
   # minicom -s
   ```

   a. Select `Serial port setup` from the `[configuration]` menu.
   b. In the setup menu, type `A` to change the `Serial Device` setting.

      If the `Serial Device` setting is `/dev/modem`, then change it to `/dev/ttyS0`.
   c. Press **<ENTER>** to confirm the change.
   d. Press **<ENTER>** again to exit the setup menu.
   e. Select the **Save setup as dfl** menu option.
   f. Select the **Exit** menu option.

      This initializes the serial port.
   g. Type **<CONTROL>-a q** to finally exit `minicom`.
5. Follow a similar configuration procedure with the other computer connected to the serial cable.

## C.2 OS-Level Testing

The serial connection between the two computers can be tested with the `minicom` serial communications program.

1. Remotely login to each computer.
2. Launch the `minicom`(1) serial communications program on each computer.
3. Type some text into a `minicom` window.
4. Type **<CONTROL>-a q** to finally exit `minicom`.

This should determine that the serial connection is correct.