J2ME Building Blocks for Mobile Devices

White Paper on KVM and the Connected, Limited Device Configuration (CLDC)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

May 19, 2000

Contents

1

Executive Summary

Information Appliances and the Wireless Revolution

Connected, personalized, intelligent information appliances are becoming increasingly important in our business and private lives. These appliances, which include devices such as cell phones, two-way pagers, personal organizers, screen phones, and POS terminals, have many things in common. But they are also diverse in features, form, and function. They tend to be special-purpose, limited-function devices, not the general-purpose computing machines we have known in the past.

The number of these information appliances is increasing rapidly. For instance, the total number of cell phone shipments is expected to be around 350 million units this year alone. The total number of wireless subscribers in the world is expected to exceed one billion by the end of 2002 or early 2003. Compare this to the installed base of personal computers, which at the beginning of 2000 was around 311 million worldwide.

Everything Connected

We anticipate that within the next two to five years, the majority of new information appliances will be connected to the Internet. This will lead to a radical change in the way people perceive and use these devices. The users of the information appliances will want to access information—Web content, enterprise data, and personal data—conveniently from anywhere, any time, and from a variety of devices (Figure 1-1).

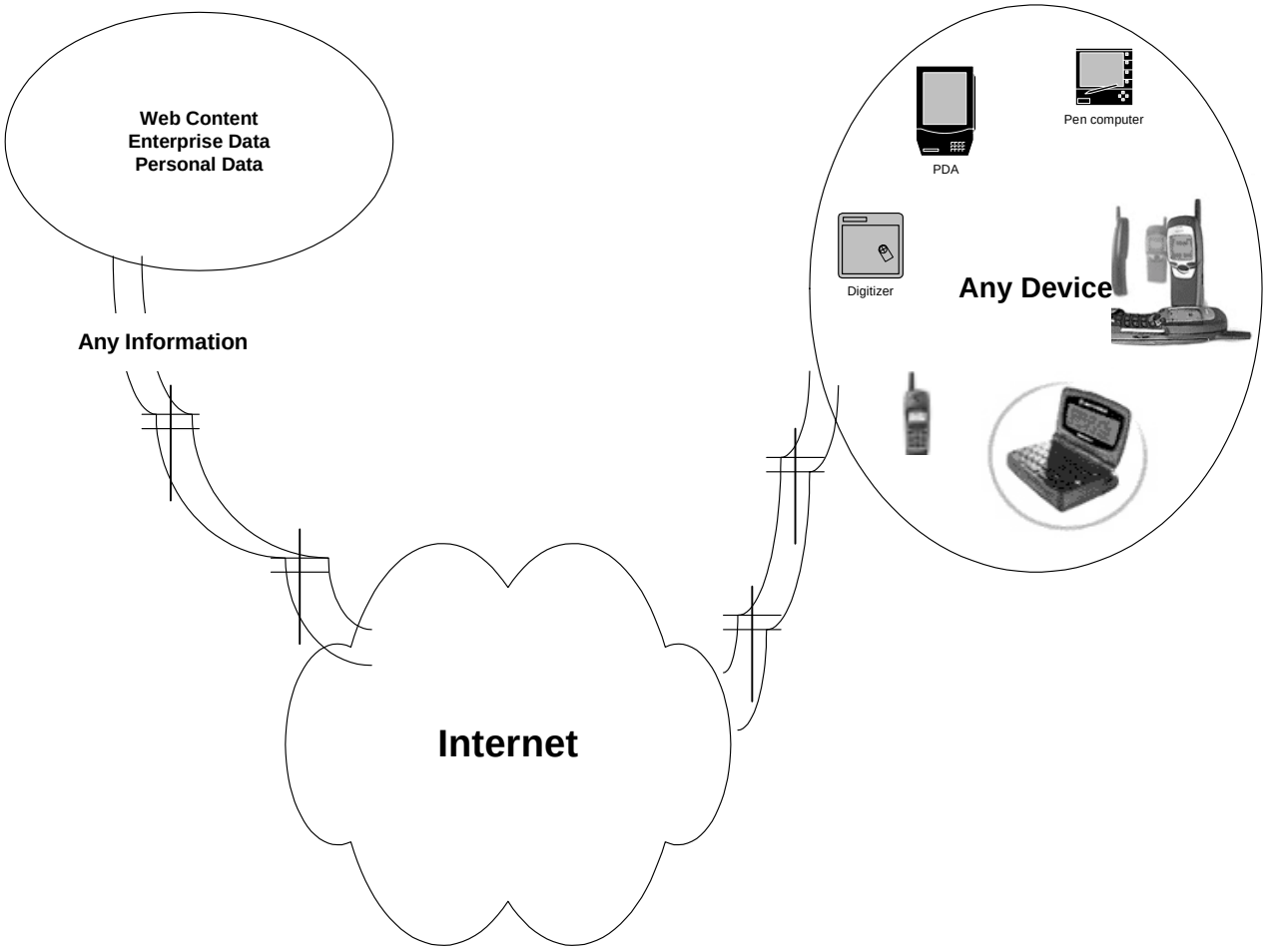

Figure 1-1 Everything connected to the Internet

Customizable, Personal Services

An important consequence of the connected nature of new information appliances is that these devices will be much more customizable and personal than the appliances we have today.

Unlike in the past, when devices such as cell phones came with a hard-coded feature set, the new devices will allow the users to customize their devices by downloading new services and applications from the Internet.

Several wireless device manufacturers are already working on cell phones that allow the users to download new applications such as interactive games, banking and ticketing applications, wireless collaboration and so on (Figure 1-2).

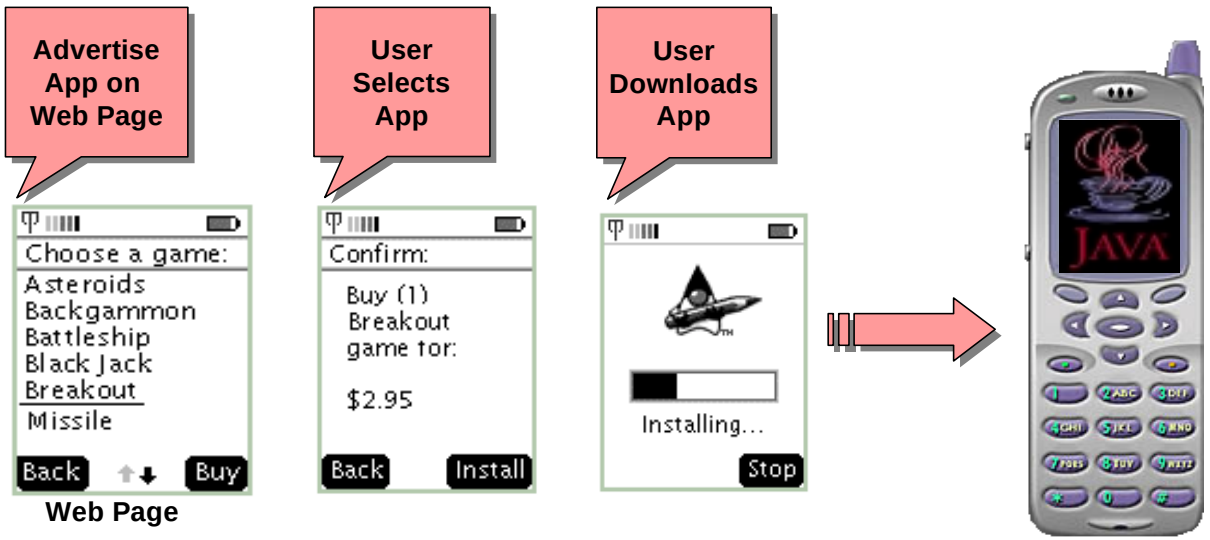

Figure 1-2 Downloading customized services

Such customizability will not be limited to just communication devices such as cell phones or two-way pagers. For instance, it is quite realistic to imagine automobile engines to obtain new service updates as they become available, washing machines to download new washing programs dynamically, electronic toys to download updated game programs, and so on.

The need for customizability and personalized applications requires a lot more from the application development platform than is available in mainstream small consumer devices today. With the power of a widely used, extensible programming platform such as the Java™ platform, the development of such applications and services will become significantly easier.

Java™ 2 Platform Micro Edition (J2ME™)

To meet the demand for information appliances in the rapidly developing consumer and embedded markets, Sun has extended the scope of Java technology with the introduction of *Java™ 2 Platform, Micro Edition (J2ME™)*. The versatility of the Java application development environment is now enabling the development of many new and powerful information appliance products. Java technology enables users, service providers, and device manufacturers to take advantage of a rich portfolio of application content that can be delivered to the user's device on demand, by wired or wireless connections.

The main benefits of CLDC devices involve:

- Cross-Platform

  Work is transferred between CLDC and other devices.

- Dynamic Content

  Content is determined by user experience, and information transfer between CLDC and other devices.

- Security

- Developer Community

  The developer talent needed for these devices already exists and is readily available for CLDC devices.

J2ME Configurations and Profiles

Serving the information appliance market calls for a large measure of flexibility in how computing technology and applications are deployed. This flexibility is required because of

1. the large range of existing device types and hardware configurations,
2. constantly improving device technology,
3. the diverse range of existing applications and features, and
4. the need for applications and capabilities to change and grow (often in unforeseen ways) in order to accommodate the future needs of the consumer.

Users want the ability to purchase economically-priced products with basic functionality and then use them with ever-increasing sophistication.

In order to support this kind of flexibility and customizable deployment demanded by the consumer and embedded market, the J2ME architecture is designed to be modular and scalable. This modularity and scalability are defined by J2ME as three layers of software built upon the *Host Operating System* of the device:

- *Java Virtual Machine*. This layer is an implementation of a Java virtual machine that is customized for a particular device's host operating system and supports a particular J2ME configuration.
- *Configuration*. The configuration is less visible to users, but is very important to profile implementers. It defines the minimum set of Java virtual machine features and Java class libraries available on a particular "category" of devices representing a particular "horizontal" market segment. In a way, a configuration defines the "lowest common denominator" of the Java platform features and libraries that the developers can assume to be available on all devices.
- *Profile*. The profile is the most visible layer to users and application providers. It defines the minimum set of Application Programming Interfaces (APIs) available on a particular "family" of devices representing a particular "vertical" market segment. Profiles are implemented "upon" a particular configuration. Applications are written "for" a particular profile and are thus portable to any device that "supports" that profile. A device can support multiple profiles.

The three layers built upon the Host Operating System are illustrated in Figure 1-3.

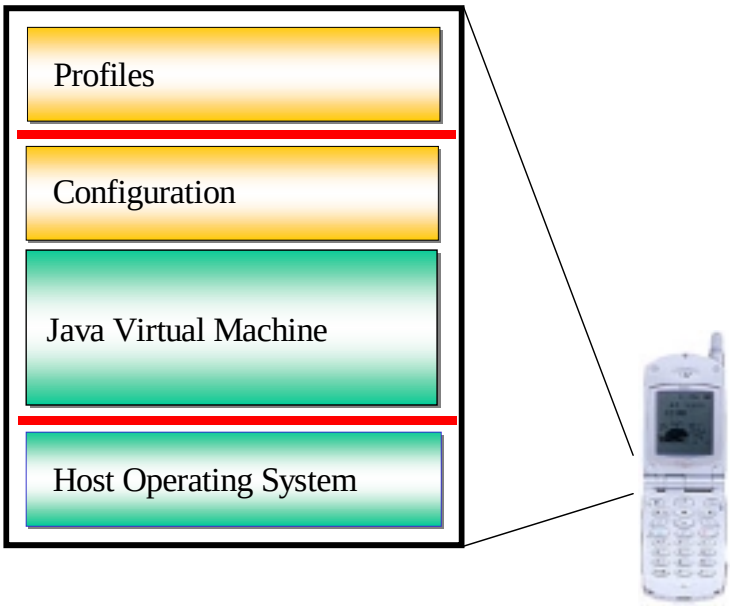

Figure 1-3 J2ME software layer stack

In J2ME, a Java virtual machine implementation and a configuration specification are very closely aligned. Together they are designed to capture just the essential capabilities of each category of device. Further differentiation into device families is provided with the additional APIs specified at the profile layer. To meet the need of new and exciting applications, profiles can be augmented with additional Java class libraries.

Over time, as device manufacturers develop new families and/or categories of devices, J2ME will provide a range of profiles, configurations, and virtual machine technologies, each optimized for the different application requirements and memory footprints commonly found in the consumer and embedded marketplace. These will be specified through the *Java Community Process (JCP).*

The J2ME architecture currently has two configurations that have been defined using the JCP. The *Connected Device Configuration (CDC)* uses the classic Java virtual machine, a full-featured VM that includes all the functionality of a

virtual machine residing on a desktop system. This configuration is intended for devices with at least a few megabytes of available memory.

For wireless devices and other systems with severely constrained memory environments, J2ME uses the *Connected Limited Device Configuration (CLDC)*, discussed in more detail below.

Connected, Limited Device Configuration (CLDC)

The configuration for mobile devices or the Connected, Limited Device Configuration (CLDC) defines targeted Java platforms which are small, resource-constrained devices, each with a memory budget in the range of 160 kB to 512 kB. The CLDC is composed of the K Virtual Machine (KVM) and core class libraries that can be used on a variety of devices such as cell phones, two-way pagers, personal organizers, home appliances, and so on. Eighteen companies, mostly wireless device manufacturers, have participated in the definition of this configuration using the Java Community Process (JCP).

The K Virtual Machine

The K Virtual Machine (KVM), a key feature of the J2ME architecture, is a highly portable Java virtual machine designed from the ground up for small-memory, limited-resource, network-connected devices such as cellular phones, pagers, and personal organizers. These devices typically contain 16- or 32-bit processors and a minimum total memory footprint of approximately 128 kilobytes. However, the KVM can be deployed flexibly in a wide variety of devices appropriate for various industries and the large range of trade-offs among processor power, memory size, device characteristics, and application functionality they engender.

About this White Paper

The purpose of this white paper is to describe the current reference implementation of KVM along with the closely related Connected, Limited Device Configuration (CLDC). Chapter 2 sets the stage for this discussion by providing an expanded introduction to the Java 2 Micro Edition. Then, Chapter 3 reviews the essential features of the Connected, Limited Device Configuration and the APIs that it defines. Chapter 4 provides more detailed information on KVM and on what is required when porting it to new devices.

Finally, Chapter 5 briefly discusses the future directions of KVM and CLDC technology.

2

Introduction to the Java 2 Platform Micro Edition, CLDC, and KVM

Java Editions

Recognizing that one size does not fit all, Sun has grouped its Java technologies into three editions, each aimed at a specific area of today's vast computing industry:

- *Java 2 Enterprise Edition (J2EE)*—for enterprises needing to serve their customers, suppliers, and employees with solid, complete, and scalable Internet business server solutions.
- *Java 2 Standard Edition (J2SE)*—for the familiar and well-established desktop computer market.
- *Java 2 Micro Edition (J2ME)*—for the combined needs of:
  — consumer and embedded device manufacturers who build a diversity of information devices;
  — service providers who wish to deliver content to their customers over those devices; and
  — content creators who want to make compelling content for small, resource-constrained devices.

Each Java edition defines a set of technology and tools that can be used with a particular product:

- Java virtual machines that fit inside a wide range of computing devices;
- libraries and APIs specialized for each kind of computing device; and
- tools for deployment and device configuration.

Figure 2-1 below illustrates the target markets of each edition.

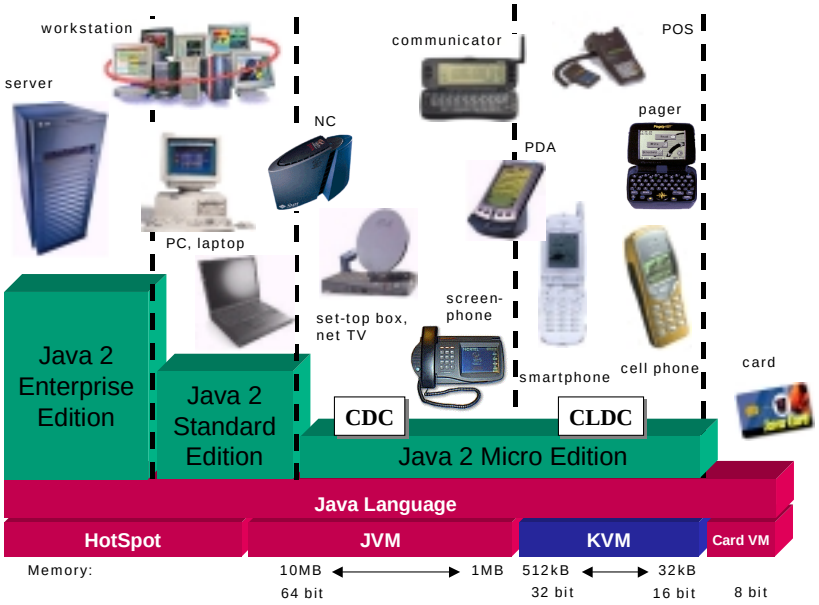

Figure 2-1 Java 2 editions and their target markets

Java 2 Platform Micro Edition (J2ME)

J2ME specifically addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box, an appliance almost as powerful as a desktop computer. Like the “larger” Java editions, Java 2 Micro Edition maintains the qualities that Java technology has become known for:

- built-in consistency across products in terms of running anywhere, any time, on any device;
- the power of a high-level object-oriented programming language with a large developer base;
- portability of code;
- safe network delivery; and
- upward scalability with J2SE and J2EE.

With J2ME, Sun provides a complete end-to-end solution for creating dynamically extensible, networked products and applications for the consumer and embedded market. J2ME enables device manufacturers, service providers, and content creators to gain a competitive advantage and capitalize on new revenue streams by developing and deploying compelling new applications and services to their customers worldwide.

At a high level, J2ME is currently targeted at two broad categories of products:

- *Shared, fixed, connected information devices*. In Figure 2-1, this category is represented by the grouping labeled CDC (Connected Device Configuration). Typical examples of devices in this category include TV set-top boxes, Internet TVs, Internet-enabled screenphones, high-end communicators, and automobile entertainment/navigation systems. These devices have a large range of user interface capabilities, memory budgets in the range of 2 to 16 megabytes, and persistent, high-bandwidth network connections, most often using TCP/IP.
- *Personal, mobile, connected information devices*. In Figure 2-1, this category is represented by the grouping labeled CLDC (Connected, Limited Device Configuration). Cell phones, pagers and personal organizers are examples of devices in this category. These devices have very simple user interfaces (compared to desktop computer systems), minimum memory budgets starting at about 128 kilobytes, and low bandwidth, intermittent network connections. In this category of products, network communications are often not based on the TCP/IP protocol suite.

The line between these two product categories is fuzzy and becoming more so every day. As a result of the ongoing technological convergence in the computer, telecommunication, consumer electronics and entertainment industries, there will be less distinction between general-purpose computers, personal communication devices, consumer electronics devices and entertainment devices. Also, future devices are more likely to use wireless connectivity instead of traditional fixed or wired networks. In practice, the line between the two categories is defined more by the memory budget, bandwidth considerations, battery power consumption, and physical screen size of the device, rather than by its specific functionality or type of connectivity.

J2ME Building Blocks: Configurations and Profiles

While connected consumer devices such as cell phones, pagers, personal organizers and set-top boxes have many things in common, they are also diverse in form, function and features. Information appliances tend to be

special-purpose, limited-function devices. To address this diversity, an essential requirement for J2ME is not only small size but also modularity and customizability.

The J2ME architecture is modular and scalable so that it can support the kinds of flexible deployment demanded by the consumer and embedded markets. To enable this, J2ME provides a range of virtual machine technologies, each optimized for the different processor types and memory footprints commonly found in the consumer and embedded marketplace.

For low-end, resource-limited products, J2ME supports minimal configurations of the Java virtual machine and Java APIs that embody just the essential capabilities of each kind of device. As device manufacturers develop new features in their devices, or service providers develop new and exciting applications, these minimal configurations can be expanded with additional APIs or with a richer complement of Java virtual machine features. To support this kind of customizability and extensibility, two essential concepts are defined by J2ME:

- *Configuration*. A J2ME configuration defines a minimum platform for a "horizontal" category or grouping of devices, each with similar requirements on total memory budget and processing power. A configuration defines the Java language and virtual machine features and minimum class libraries that a device manufacturer or a content provider can expect to be available on all devices of the same category.
- *Profile*. A J2ME profile is layered on top of (and thus extends) a configuration. A profile addresses the specific demands of a certain "vertical" market segment or device family. The main goal of a profile is to guarantee interoperability within a certain vertical device family or domain by defining a standard Java platform for that market. Profiles typically include class libraries that are far more domain-specific than the class libraries provided in a configuration.

J2ME configurations and profiles are defined through the *Java Community Process (JCP)*.

J2ME Profiles

Application portability is a key benefit of Java technology in the desktop and enterprise server markets. Portability is also a critical element of the J2ME value proposition for consumer devices. However, application portability requirements in the consumer space are generally quite different from

portability requirements demanded by the desktop and server markets. In most cases consumer devices have substantial differences in memory size, networking, and user interface capabilities, making it very difficult to support all devices with just one solution.

In general, the consumer device market is not so homogeneous that end users expect or require universal application portability. Rather, in the consumer space, applications should ideally be fully portable between devices of the same kind. For example, consider the following types of consumer devices:

- cellular telephones
- washing machines
- intercommunicating electronic toys

It seems clear that each of these represents a different “market segment” or “device family” or “application domain.” As such, consumers would expect useful applications to be portable within a device family. For example:

- I would expect my discount broker’s stock trading application to work on each of my cell phones, even though they are from different manufacturers.
- If I found a wonderful grape-juice-stain-removing wash cycle application on the Internet, I would be annoyed if it ran on my old brand-X washer, but not my new brand-Z washer.
- My child’s birthday party would be less enjoyable if the new robot doesn’t “talk to” and “play games with” the new electronic teddy bear.

On the other hand, consumers don’t expect the stock application or an automobile service program to run on the washing machine or the toy robot. In other words, application portability *across* different device categories is not necessarily very important.

In addition, there are important economic reasons to keep these device families separate. Consumer devices compete heavily on cost and convenience, and these factors often translate directly into limitations on physical size and weight, processor power, memory size, and power consumption (in battery-powered devices.) Consumers’ wallets will always favor devices that perform the desired functions, but that do not have added cost for unnecessary features.

Thus, the J2ME framework provides the concept of a *profile* to make it possible to define Java platforms for specific vertical markets. A profile defines a Java platform for a specific vertical market segment or device category. Profiles can serve two distinct portability requirements:

- A profile provides a complete toolkit for implementing applications for a particular kind of device, such as a pager, set-top box, cell phone, washing machine, or interactive electronic toy.
- A profile may also be created to support a significant, coherent group of applications that might be hosted on several categories of devices. For example, while the differences between set-top boxes, pagers, cell phones, and washing machines are significant enough to justify creating a separate profile for each, it might be useful for certain kinds of personal information management or home banking applications to be portable to each of these devices. This could be accomplished by creating a separate profile for these kinds of applications and ensuring that this new profile can be easily and effectively supported on each of the target devices along with its "normal" more device-specific profile.

It is possible for a single device to support several profiles. Some of these profiles will be very device-specific, while others will be more application-specific. Applications are written "for" a specific profile and are required to use only the features defined by that profile. Manufacturers choose which profile(s) to support on each of their devices, but are required to implement all features of the chosen profile(s). The value proposition to the consumer is that any application written for a particular profile will run on any device that supports that profile.

In its simplest terms, a profile is a contract between an application and a J2ME vertical market segment. All the devices in the market segment agree to implement all the features defined in the profile. And the application agrees to use only those features defined in the profile. Thus, portability is achieved between the applications and the devices served by that profile. New devices can take advantage of a large and familiar application base. Most importantly new, compelling applications (perhaps completely unforeseen by the original profile designers) can be dynamically downloaded to existing devices.

At the implementation level, a profile is defined simply as a collection of Java APIs and class libraries that reside on top of a specified configuration and that provide the additional domain-specific capabilities for devices in a specific market segment.

In our example above, each of the three families of devices (cell phones, washing machines, and intercommunicating toys) would be addressed by a separate J2ME profile. Of course, the only one of these profiles in existence at the current time is the MIDP, designed for cell phones and related devices.

Profiles and the specific rules for defining J2ME profiles are described in more detail in separate specifications.

J2ME Configurations

In J2ME, an application is written “for” a particular profile, and a profile is “based upon” or “extends” a particular configuration. Thus, all of the features of a configuration are automatically included in the profile and may be used by applications written for that profile.

A configuration defines a Java platform for a “horizontal” category or grouping of devices with similar requirements on total memory budget and other hardware capabilities. More specifically, a configuration:

- specifies the Java programming language features supported,
- specifies the Java virtual machine features supported,
- specifies the basic Java libraries and APIs supported.

J2ME is designed so that it can be deployed in more than one configuration. Each configuration specifies the Java virtual machine features and a set of APIs that the profile implementer (and the applications using that profile) can safely assume to be present on all devices when shipped from the factory. Profile implementers must design their code to stay within the bounds of the Java virtual machine features and APIs specified by that configuration.

In its simplest terms, a configuration is a contract between a profile implementer and a device’s Java virtual machine. The virtual machines of all the devices in the market segment agree to implement all the features defined in the configuration. And the profile implementers agree to use only those features defined in the configuration. Thus, portability is achieved between the profile and the devices served by that configuration. New devices can take advantage of existing profiles. And new profiles can be installed on existing devices.

In our example above, each of the three profiles (for cell phones, washing machines, and intercommunicating toys) would most likely be built upon the same configuration, the CLDC. This configuration provides all the basic functionality to serve the needs of each of these, and perhaps many more, profiles.

To avoid fragmentation, there will be a very limited number of J2ME configurations. Currently, the goal is to define two standard J2ME configurations (see Figure 2-2):

- **Connected, Limited Device Configuration (CLDC)**. The market consisting of personal, mobile, connected information devices is served by the CLDC. This configuration includes some new classes, not drawn from the J2SE APIs, designed specifically to fit the needs of small-footprint devices.
- **Connected Device Configuration (CDC)**. The market consisting of shared, fixed, connected information devices is served by the Connected Device Configuration (CDC). To ensure upward compatibility between configurations, the CDC shall be a superset of the CLDC.

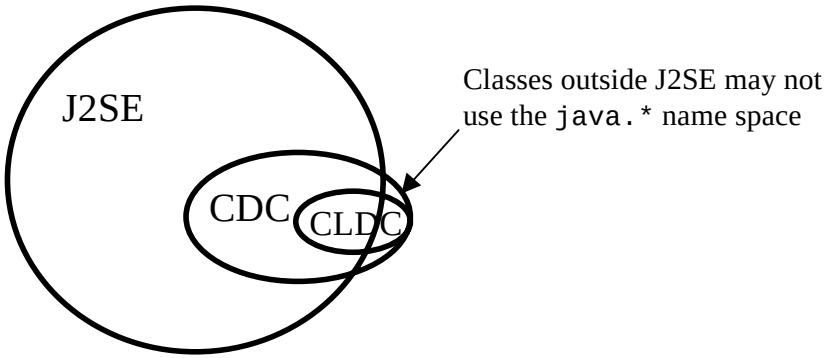

Figure 2-2 Relationship between J2ME configurations and Java 2 Standard Edition

Figure 2-2 illustrates the relationship between CLDC, CDC and Java 2 Standard Edition (J2SE). As shown in the figure, the majority of functionality in CLDC and CDC has been inherited from J2SE. Each class inherited from J2SE must be precisely the same or a subset of the corresponding class in Java 2 Standard Edition. In addition, CLDC and CDC may introduce a number of features, not drawn from the J2SE, designed specifically to fit the needs of small-footprint devices. For further details, refer to *Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)*, Sun Microsystems, Inc.

The most important reason for the configuration layer of J2ME is that configurations and Java virtual machines are very closely related and are rather complex pieces of software. Small differences in a configuration's specification can require a large number of modifications to the internal design of a Java virtual machine, which would be very expensive and time-consuming

to maintain. Having a small number of configurations means that a relatively small number of Java virtual machine implementations can serve the needs of both a large number of profiles and a large number different device hardware types. This economy of scale provided by J2ME is very important to the success and cost-effectiveness of devices in the consumer and embedded industry.

KVM

The KVM is a compact, portable Java virtual machine specifically designed from the ground up for small, resource-constrained devices. The high-level design goal for the KVM was to create the smallest possible "complete" Java virtual machine that would maintain all the central aspects of the Java programming language, but would run in a resource-constrained device with only a few hundred kilobytes total memory budget.

More specifically, the KVM was designed to be:

- small, with a static memory footprint of the virtual machine core in the range of 40 kilobytes to 80 kilobytes (depending on compilation options and the target platform,)
- clean, well-commented, and highly portable,
- modular and customizable,
- as "complete" and "fast" as possible without sacrificing the other design goals.

The "K" in KVM stands for "kilo." It was so named because its memory budget is measured in kilobytes (whereas desktop systems are measured in megabytes). KVM is suitable for 16/32-bit RISC/CISC microprocessors with a total memory budget of no more than a few hundred kilobytes (potentially less than 128 kilobytes). This typically applies to digital cellular phones, pagers, personal organizers, and small retail payment terminals.

The minimum total memory budget required by a KVM implementation is about 128 kB, including the virtual machine, the minimum Java class libraries specified by the configuration, and some heap space for running Java applications. A more typical implementation requires a total memory budget of 256 kB, of which half is used as heap space for applications, 40 to 80 kB is needed for the virtual machine itself, and the rest is reserved for configuration and profile class libraries. The ratio between volatile memory (e.g., DRAM) and non-volatile memory (e.g., ROM or Flash) in the total memory budget

varies considerably depending on the implementation, the device, the configuration, and the profile. A simple KVM implementation without system class prelinking support needs more volatile memory than a KVM implementation with system classes (or even applications) preloaded into the device.

The actual role of a KVM in target devices can vary significantly. In some implementations, the KVM is used on top of an existing native software stack to give the device the ability to download and run dynamic, interactive, secure Java content on the device. In other implementations, the KVM is used at a lower level to also implement the lower-level system software and applications of the device in the Java programming language. Several alternative usage models are possible.

At the present time, the KVM and CLDC are closely related. CLDC runs only on top of KVM and CLDC is the only configuration supported by KVM. However, over time it is expected that CLDC will run on other J2ME virtual machine implementations and that the KVM may perhaps support other configurations as they are defined.

For further information on the KVM, refer to the KVM web site at `http://java.sun.com/products/kvm`.

The KVM is derived from a research system called *Spotless* developed originally at Sun Microsystems Laboratories. More information on Spotless is available in the Sun Labs technical report “The Spotless System: Implementing a Java system for the Palm Connected Organizer” (Sun Labs Technical Report SMLI TR-99-73).

3

The Connected, Limited Device Configuration (CLDC)

As mentioned previously, the KVM and CLDC are very closely related. In essence, CLDC is the *specification* for a "class" of Java virtual machines that can run on the categories of devices targeted by CLDC and support the profiles layered on top of CLDC. The KVM is a particular *implementation* (currently the one and only Sun reference implementation) of a Java virtual machine meeting the CLDC specifications. Therefore, no discussion of KVM can be complete without an understanding of the CLDC requirements. This chapter briefly describes some of the CLDC specifications that affect the KVM.

CLDC Goals

- To define a standard Java platform for small, resource-constrained, connected devices.
- To allow dynamic delivery of Java applications and content to those devices.
- To enable 3rd party application developers to easily create applications and content that can be deployed to those devices.

CLDC Requirements

- To run on a wide variety of small devices ranging from wireless communication devices such as cellular telephones and two-way pagers to personal organizers, point-of-sale terminals and even home appliances.
- To make minimal assumptions about the native system software available in CLDC devices.
- To define a minimum complement or the “lowest common denominator” of Java technology applicable to a wide variety of mobile devices.
- To guarantee portability and interoperability of profile-level code between the various kinds of mobile (CLDC) devices.

The entire CLDC implementation (static size of the virtual machine + libraries) should fit in less than 128 kilobytes. The CLDC Specification assumes that applications can be run in as little as 32 kilobytes of Java heap space.

CLDC Scope

The CLDC configuration addresses the following areas:

- Java language and virtual machine features
- Core Java libraries (`java.lang.*`, `java.util.*`)
- Input/output
- Networking
- Security
- Internationalization

The CLDC configuration does ***not*** address the following areas. These features are addressed by profiles implemented on top of the CLDC:

- Application life-cycle management (application installation, launching, deletion)
- User interface
- Event handling
- High-level application model (the interaction between the user and the application)

Security

The CLDC specification addresses the following topics related to security:

- Low-level-virtual machine security is achieved by requiring downloaded Java classes to pass a classfile verification step.
- Applications are protected from each other by being run in a closed "sandbox" environment.
- Classes in protected system packages cannot be overridden by applications.

Adherence to the Java Language Specification

The general goal for a Java VM supporting CLDC is to be as compliant with the *Java™ Language Specification* as is feasible within the strict memory limits of the target devices. Except for the following differences, a Java VM supporting CLDC shall be compatible with Chapters 1 through 17 of *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1:

- No support for floating point data types (`float` and `double`).
- No support for finalization of class instances. The method `Object.finalize()` does not exist.
- Limitations on error handling. Most subclasses of `lava.lang.Error` are not supported. Errors of these types are handled in an implementation-dependent manner appropriate for the device (in contrast, CLDC includes a fairly complete set of *exception* classes.)

Adherence to the Java Virtual Machine Specification

The general goal for a Java VM supporting CLDC is to be as compliant with the *Java™ Virtual Machine Specification* as is possible within strict memory constraints. Except for the following differences, a Java VM supporting CLDC shall be compatible with the Java Virtual Machine as specified in the *The Java Virtual Machine Specification (Java Series)* by Tim Lindholm and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X.

- No support for floating point data types (`float` and `double`).
- No support for the Java Native Interface (JNI).
- No user-defined, Java-level class loaders.
- No reflection features.
- No support for thread groups or daemon threads.
- No support for finalization of class instances.
- No weak references.
- Limitations on error handling.

Apart from floating point support, which has been omitted primarily because the majority of the CLDC target devices do not have hardware support for floating point arithmetic, the features above have been eliminated either because of:

- strict memory limitations, or
- because of potential security concerns in the absence of the full J2SE security model.

Classfile Verification

CLDC requires that a Java VM be able to identify and reject invalid classfiles. However, since the standard classfile verification approach defined by J2SE is too memory-consuming for small devices, CLDC defines an alternative mechanism for classfile verification.

In this alternative, each method in a downloaded Java classfile contains a “stackmap” attribute. This attribute is newly-defined in CLDC and is not defined by *The Java Virtual Machine Specification*. Typically, this attribute is added to standard classfiles by a “pre-verification” tool that analyzes each method in the classfile. Pre-verification is typically performed on a server or

desktop system before the classfile is downloaded to the device (see Figure 3-1). The stack map attribute increases the size of a classfile by approximately 5%.

The presence of this attribute enables a CLDC-compliant Java VM to verify Java classfiles much more quickly and with substantially less VM code and dynamic RAM consumption than the standard Java VM verification step, but with the same level of security.

Note that since stack maps have been implemented by utilizing the extensible attribute mechanism built in Java classfiles, classfiles containing stack maps will run unmodified in larger Java environments such as J2SE or J2EE.

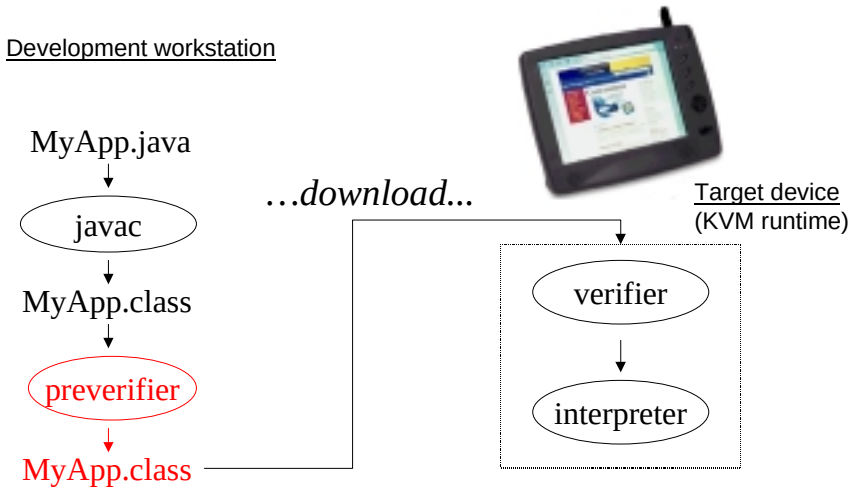

Figure 3-1 Classfile verification in CLDC/KVM

Classfile Format

In order to enable dynamic downloading of 3rd party applications and content, CLDC requires that implementations support the distribution of Java applications via compressed Java Archive (JAR) files. Whenever a Java application intended for a CLDC device is “represented publicly” or “distributed publicly” it must be formatted in a compressed Java Archive (JAR) file, and classfiles within a JAR file must contain the stackmap attribute. (However, once an application is admitted into the kinds of closed, private,

vendor-controlled networks sometimes used with today's information appliances, the vendor is free to use a different format.)

CLDC Libraries

In order to ensure upward compatibility and portability of applications, the majority of the class libraries included in CLDC are a subset of those specified for the larger Java editions (J2SE and J2EE). Only those classes that are appropriate for mobile devices are specified by CLDC.

Classes Inherited from J2SE

The following classes have been inherited directly from Java 2 Standard Edition. Each class is a subset of the corresponding class in J2SE. The methods and fields of these classes are a subset of the complete classes as defined in the larger Java editions. Only those methods and fields that are appropriate for "connected, limited devices" are specified by CLDC.

System Classes

From `java.lang`:

```
Object, Class, Runtime, System, Thread, Runnable,
String, StringBuffer, Throwable
```

Data Type Classes

From `java.lang`:

```
Boolean, Byte, Short, Integer, Long, Character
```

Collection Classes

From `java.util`:

```
Vector, Stack, Hashtable, Enumeration
```

I/O Classes

From `java.io`:

`InputStream, OutputStream, ByteArrayInputStream, ByteArrayOutputStream, DataInput, DataOutput, DataInputStream, DataOutputStream, Reader, Writer, InputStreamReader, OutputStreamWriter, PrintStream`

Calendar and Time Classes

From `java.util`:

`Calendar, Date, TimeZone`

Additional Utility Classes

`java.util.Random, java.lang.Math`

Exception Classes

From `java.lang`:

`Exception, ClassNotFoundException, IllegalAccessException, InstantiationException, InterruptedException, RuntimeException, ArithmeticException, ArrayStoreException, ClassCastException, IllegalArgumentException, IllegalThreadStateException, NumberFormatException, IllegalMonitorStateException, IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, SecurityException`

From `java.util`:

`EmptyStackException, NoSuchElementException`

From `java.io`:

`EOFException, IOException, InterruptedException, UnsupportedEncodingException, UTFDataFormatException`

Error Classes

From `java.lang`:

```
Error, VirtualMachineError, OutOfMemoryError
```

Limitations

- CLDC includes limited support for the translation of Unicode characters to and from a sequence of bytes using Readers and Writers.
- CLDC does not support class `java.util.Properties`, which is part of J2SE. However, a limited set of properties beginning with the keyword "`microedition`" can be accessed by calling the method `System.getProperty(String key)`.

CLDC-Specific Classes

The J2SE and J2EE libraries provide a rich set of functionality for handling input and output access to storage and networking systems via the `java.io` and `java.net.*` packages, However, it is difficult to make all this functionality fit in a small device with only a few hundred kilobytes of total memory budget.

This has led to a generalization of the J2SE network and I/O classes for J2ME. The general goal for this new system is to be a precise functional subset of J2SE classes, which can easily map to common low-level hardware or to any J2SE implementation, but with better extensibility, flexibility and coherence in supporting new devices and protocols.

Instead of using a collection of totally different kinds of abstractions for different forms of communication, a set of related abstractions are used at the application programming level.

General Form

All connections are created using a single static method in a system class called `javax.microedition.Connector`. If successful, this method will return an object that implements one of the generic connection interfaces. There are number of these interfaces that form a hierarchy with the `Connection` interface being the root. The method takes a URL parameter in the general form:

```
Connector.open("<protocol>:<address>;<parameters>");
```

Examples

NOTE—These examples are provided for illustration only. CLDC itself does not define any protocol implementations. It is not expected that a particular J2ME profile would provide support for all these kinds of connections. J2ME profiles may also support protocols not shown below.

- HTTP records: `Connector.open("http://www.foo.com");`
- Sockets: `Connector.open("socket://129.144.111.222:9000");`
- Communication ports: `Connector.open("comm:0;baudrate=9600");`
- Datagrams: `Connector.open("datagram://129.144.111.333");`
- Files: `Connector.open("file:foo.dat");`
- Network file systems: `Connector.open("nfs:/foo.com/foo.dat");`

Generic Connection Framework Interfaces

This new framework is implemented using a hierarchy of `Connection` interfaces that group together classes of protocols with the same semantics. This hierarchy consists of the following seven interfaces from `javax.microedition.io`:

```
Connection, InputConnection, OutputConnection,
StreamConnection, ContentConnection, DatagramConnection,
StreamConnectionNotifier
```

4

The K Virtual Machine (KVM)

Introduction to the KVM

The KVM (also known as the K Virtual Machine) is a compact, portable Java virtual machine intended for small, resource-constrained devices such as cellular phones, pagers, personal organizers, mobile Internet devices, point-of-sale terminals, home appliances, and so forth.

The high-level design goal for the KVM was to create the smallest possible "complete" Java virtual machine that would maintain all the central aspects of the Java programming language, and that would nevertheless run in a resource-constrained device with only a few tens or hundreds of kilobytes of available memory (hence the name K, for kilobytes). More specifically, the KVM is designed to be:

- Small, with a static memory footprint of the virtual machine core in the range 40 kilobytes to 80 kilobytes (depending on the target platform and compilation options).
- Clean and highly portable.
- Modular and customizable.
- As "complete" and "fast" as possible without sacrificing the other design goals.

The KVM is implemented in the C programming language, so it can easily be ported onto various platforms for which a C compiler is available. The virtual machine has been built around a straightforward bytecode interpreter with

various compile-time flags and options to aid porting efforts and improve space optimization.

The following sections provide highlights of the KVM reference implementation. Additional detail can be found in the *KVM Porting Guide*.

Sun Implementations

Sun's KVM reference implementation can be compiled and tested on two platforms:

- Solaris
- Windows

A third implementation can be compiled and used on:

- Palm OS

The Solaris and Windows platforms are used for KVM development, debugging, testing, and demonstration. They leverage a wealth of development tools and allow rapid porting and development efforts due to the increased workstation-class performance. The Solaris and Windows versions of the KVM are used as the basis for official CLDC reference implementations that customers can use for the device-specific ports.

The Palm OS platform is the primary test bed to ensure that KVM meets its goals of efficiently executing on a resource-limited device.

Other Implementations

At the time of this writing, the KVM has been successfully ported to more than 25 devices by Sun's Early Access licensees.

Compiler Requirements

The KVM is designed to be built with any C compiler capable of compiling ANSI-compliant C files. The only non-ANSI feature in the source code is its use of 64-bit integer arithmetic.

Our reference implementation has only been tested on machines with 32-bit pointers and that do not require "far" pointers of any sort. We do not know if it will run successfully on platforms with pointers of other sizes.

The codebase has been successfully compiled with the following compilers:

- Metrowerks CodeWarrior Release 6 for Palm,
- Sun DevPro C Compiler 4.2 on Solaris,
- GNU C compiler on Solaris,
- Microsoft Visual C++ 6.0 Professional on Windows 98 and Windows NT 4.0.

Porting KVM

The majority of KVM source code is common to all implementations. The relatively small amount of machine-dependent and/or platform-specific code is isolated to small number of files. New or modified versions of these files must be created for each port.

A relatively small number of well-specified runtime functions must be implemented in order to provide the necessary interface between KVM and the underlying native operating environment for such operations as:

- Initializations
- Finalizations (clean-up)
- Heap allocation/deallocation
- Fatal error reporting
- Event handling
- Current time

Compilation Control

A large number of macro definitions are provided to control features relating to:

- Data alignment
- Long (64-bit) integers
- Floating point (if used)
- Endianness (big endian vs. little endian)
- Classpaths (if used or not)
- System class preloading (ROMizing)
- Platform-specific features
- Memory allocation
- Garbage collection
- Interpreter options and optimizations
- Debugging and tracing options
- Networking and storage options (Generic Connections)

Virtual Machine Startup and JAM

On desktop implementations, the KVM can be run from the command line, as is done with J2SE.

On devices with user interface capable of launching native applications (such as Palm OS) the KVM can be configured to run in that fashion.

For devices that do not have such a user interface, the KVM provides a reference implementation of a facility called the *Java Application Manager* (JAM), which serves as an interface between the host operating system and the virtual machine. The JAM assumes that applications are available for downloading as JAR files by using a network (typically HTTP) or storage protocol implemented using the Generic Connection framework. The JAM reads the contents of the JAR file and an associated descriptor file from the Internet, and launches the KVM with the main class as a parameter.

For development and testing purposes, desktop implementations of the KVM can be configured to use the JAM as an alternative startup strategy.

Class Loading

The KVM reference implementation can load classes from a directory path as well as from a JAR file. Alternative device-specific class loading mechanisms can be created where necessary.

64-Bit Support

The KVM is most easily ported to compilers that support 64-bit arithmetic. However, macros are provided that can be redefined to perform the appropriate operations for compilers that do not support 64-bit integers.

Native Code

The KVM does not support the Java Native Interface (JNI). Rather, any native code called from the virtual machine must be linked directly into the virtual machine at compile time. Invoking native methods is accomplished via native method lookup tables, which must be created during the build process.

Macros are provided to make the implementation of native methods as easy and error-free as possible. However, native methods are inherently complex, and the consequences of mistakes are frequently severe. We advise studying the KVM Porting Guide with great care before attempting to write your own native methods.

Event Handling

For porting flexibility, there are four ways in which notification and handling of events can be done in the KVM:

- Synchronous notification (blocking).
- Polling in Java code.
- Polling in the bytecode interpreter.
- Asynchronous notification.

Each KVM port can use the mechanism that is most appropriate for its platform.

Classfile Verification

As described in the CLDC chapter, KVM makes use of the new “stack map” method attribute in order to quickly and efficiently verify classfiles.

A pre-verification tool written in C is supplied with the KVM reference implementation. This tool can be compiled and run on Solaris and Windows.

Java Code Compact (ROMizer)

The KVM supports the *JavaCodeCompact* (JCC) utility (also known as the class *prelinker, preloader* or *ROMizer*). This utility allows Java classes to be linked directly in the virtual machine, reducing VM startup time considerably.

At the implementation level, the JavaCodeCompact utility combines Java class files and produces a C file that can be compiled and linked with the Java virtual machine.

In conventional class loading, you use `javac` to compile Java source files into Java class files. These class files are loaded into a Java system, either individually, or as part of a jar archive file. Upon demand, the class loading mechanism resolves references to other class definitions.

JavaCodeCompact provides an alternative means of program linking and symbol resolution, one that provides a less-flexible model of program building, but that helps reduce the VM’s bandwidth and memory requirements.

JavaCodeCompact can:

- Combine multiple input files.
- Determine an object instance’s layout and size.
- Load only designated class members, discarding others.

The JCC tool itself is written in Java, and so is portable to various development platforms.

5

Future Directions

As we move into the future, our goals for the CLDC and KVM technologies are to further evolve them, and to provide tools to accomplish the following goals:

- Optimize the class file format to reduce space requirements and to reduce time to install applications on resource-constrained devices.
- Provide better support for Java-level debugging and IDE integration.
- Improve the performance of essential virtual machine components such as the garbage collector, class loader, and thread synchronization operations.
- Provide other space and performance optimizations.