KVM Debug Wire Protocol (KDWP)

Specification, Version 1.0

Java 2 Platform Micro Edition

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

KDWP Specification, Version 1.0
February 26, 2001

901 San Antonio Road, Palo Alto, CA 94303 USA

Contents

Preface v

Who Should Use This Specification v

Related Literature v

Acknowledgements vi

1. **Introduction** 7
   - 1.1 Architectural Overview 8
   - 1.2 KDWP Packets 9
   - 1.3 Command and Reply Packet Fields 10
   - 1.4 Protocol details 14
2. `VirtualMachine` **Command Set** 17
3. `ReferenceType` **Command Set** 21
4. `ClassType` **Command Set** 23
5. `ObjectReference` **Command Set** 25
6. `StringReference` **Command Set** 27
7. `ThreadReference` **Command Set** 29

8. `ArrayReference` **Command Set** 35

9. `EventRequest` **Command Set** 37

10. `StackFrame` **Command Set** 41

11. `Event` **Command Set** 43

12. **Vendor Specific Command Set** 47

13. **Appendix: Constants** 49

Preface

This document, *KVM Debug Wire Protocol (KDWP) Specification*, defines the debugger interface for Java Virtual Machine implementations that are intended to be compatible with Sun's K Virtual Machine (KVM). KVM is commonly used as the underlying execution engine for the J2ME CLDC (Java™ 2 Micro Edition, Connected Limited Device Configuration) standard. The KVM Debug Wire Protocol (KDWP) is the protocol that is used for communication between a third party Java debugging environment and the K Virtual Machine.

Who Should Use This Specification

The audience for this document includes:

1. Device manufacturers who want to port the K Virtual Machine (KVM) or another J2ME CLDC Java Virtual Machine to their device and who want their device to support source-level Java debugging with Integrated Development Environments (IDEs) from third-party vendors.
2. IDE and tool vendors who wish to implement or port a Debug Agent in order to make their development environment capable of supporting source-level debugging of J2ME CLDC devices and applications.

Related Literature

The Java™ Language Specification (Java Series), Second Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3

Connected, Limited Device Configuration Specification, version 1.0, Java Community Process, Sun Microsystems, Inc. `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_030_j2melc.html`

Java 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices, A White Paper, Sun Microsystems, Inc. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`

Acknowledgements

This document was prepared in collaboration with engineers from Motorola Wireless Software, Applications and Services (WSAS) division in Austin, Texas. Many thanks to Motorola for their development efforts and support. We would also like to thank Borland, Metrowerks and Nokia for their participation and feedback.

1

Introduction

This document, *KVM Debug Wire Protocol (KDWP) Specification*, defines the debugger interface for Java Virtual Machine implementations that are intended to be compatible with Sun's K Virtual Machine (KVM). KVM is commonly used as the underlying execution engine for the J2ME CLDC (Java™ 2 Micro Edition, Connected Limited Device Configuration) standard.

The *KVM Debug Wire Protocol (KDWP)* is the protocol that is used for communication between a *Debug Agent* (DA) and a CLDC-compliant J2ME Java Virtual Machine (usually KVM).

The high-level goal of the KDWP interface is to make it possible to plug a CLDC-compliant Java Virtual Machine flexibly into a Java development and debugging environment such as Forte.

The debugging interface specified in this document is intended to be compliant with the JPDA (Java Platform Debug Architecture) specification supported by Java 2 Standard Edition (J2SE™). Further information on the JPDA architecture is available at http://java.sun.com/products/jpda/. However, due to strict memory constraints, KVM does not implement support for the JVMDI (Java Virtual Machine Debug Interface) or the full JDWP (Java Debug Wire Protocol) specifications required by JPDA. Instead, KVM implements a subset of the JDWP known as KDWP.

The KDWP interface is derived directly from the JDWP Specification (see http://java.sun.com/products/jpda/doc/jdwp-spec.html). Note that the command sets are numbered the same as the JDWP command sets and the commands in each set are numbered as per the JDWP. This allows an implementer to support more JDWP commands directly in the KVM if deemed necessary. Like JDWP, KDWP differs from many protocol specifications in that it only details format and layout, not transport.

1.1 Architectural Overview

KDWP was designed to be a strict subset of the JDWP, primarily based on the resource constraints imposed on the small devices. In order to make KVM run with a JPDA-compatible debugger IDEs without a huge memory overhead, a *Debug Agent* (also known as *debug proxy*) program is interposed between the KVM and the JPDA-compatible debugger. The Debug Agent allows many of the memory-consuming components of a JPDA-compliant debugging environment to be processed on the development workstation instead of the KVM, therefore reducing the memory overhead that the debugging interfaces have on the KVM and target devices. As obvious, the debugging interfaces can be turned off completely (at compile time) on those platforms/ports that do not need Java-level debugging support.

At the high level, the implementation of the Java-level debugging support consists of two parts:

- the actual code in the Java Virtual Machine (usually KVM) to support a subset of the JDWP, and
- the Debug Agent that performs some of the debug commands on behalf of the Java Virtual Machine.

The overall architecture for the Java-level debugging interface is illustrated in Figure 1-1. In that figure, the topmost box represents the JPDA-compliant debugging environment ("JPDA Debugger") running on a development workstation. The debugger is connected to the Debug Agent that talks to the KVM.

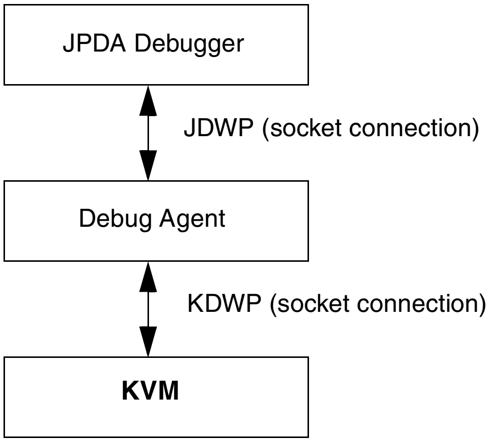

FIGURE 1-1 Java-level debugging interface architecture

The Debug Agent (DA) typically connects to the KVM via a socket connection. Similarly, the debugger connects to the Debug Agent over a socket. The debugger is unaware that it is connected to the Debug Agent. The debugger appears to be communicating directly with a JDWP-compliant Java Virtual Machine.

The KDWP protocol is designed to facilitate efficient use by a Debug Agent. Many of its abilities are tailored to that end. For instance, in some situations the Debug Agent may process the commands and issue a response directly back to the debugger without querying the Java Virtual Machine. If the command from the debugger needs data from the Java Virtual Machine, the Debug Agent communicates with the JVM via the KDWP to obtain the data. The completeness of the JDWP API that the Debug Agent provides depends on which debugger the implementer needs to support. Different debuggers may need different levels of support. The reference implementation from Sun supports a minimum set of commands that are needed by the Forte debugger.

1.2 KDWP Packets

The KDWP is packet based and is not stateful. There are two basic packet types: *command packets* and *reply packets*.

Command packets may be sent by either the DA or the target VM. They are used by the DA to request information from the target VM, or to control program execution. Command packets are sent by the target VM to notify the DA of some event in the target VM such as a breakpoint or exception.

A reply packet is sent only in response to a command packet and always provides information about the success or failure of the command. Reply packets may also carry data requested in the command (for example, the value of a field or variable). Events sent from the target VM do not require a response packet from the DA.

The KDWP is asynchronous. Multiple command packets may be sent before the first reply packet is received.

Command and reply packet headers are equal in size. This is to make transports easier to implement and abstract. The layout of each packet looks like this:

Command Packet

Header

- length (4 bytes)
- id (4 bytes)

flags (1 byte)

command set (1 byte)

command (1 byte)

data (Variable)

Reply Packet

Header

length (4 bytes)

id (4 bytes)

flags (1 byte)

error code (2 bytes)

data (Variable)

All fields and data sent via KDWP should be in big-endian format. (See the *Java™ Virtual Machine Specification* for the definition of big-endian.) The first three fields are identical in both packet types.

1.3 Command and Reply Packet Fields

Shared Header Fields

`length`

The *length* field is the size, in bytes, of the entire packet, including the length field. The header size is 11 bytes, so a packet with no data would set this field to 11.

`id`

The *id* field is used to uniquely identify each packet command/reply pair. A reply packet has the same id as the command packet to which it replies. This allows asynchronous commands and replies to be matched. The *id* field must be unique among all outstanding commands sent from one source.

(Outstanding commands originating from the debugger may use the same id as outstanding commands originating from the target VM.) Other than that, there are no requirements on the allocation of ids.

A simple monotonic counter is adequate for most implementations. It allows 2^32 unique outstanding packets and is the simplest implementation.

flags

Flags are used to alter how any command is queued and processed and to tag command packets that originate from the target VM. There is currently one flag bit defined. Future versions of the protocol may define additional flags.

0x80

Reply packet

The reply bit, when set, indicates that this packet is a reply.

Command Packet Header Fields

command set

This field is useful as a means for grouping commands in a meaningful way.

The command set space is roughly divided as follows:

0 - 63

Sets of commands sent to the target VM.

64 - 127

Sets of commands sent to the debugger/Debug Agent.

128 - 256

Vendor-defined commands and extensions.

command

This field identifies a particular command in a command set. This field, together with the command set field, is used to indicate how the command packet should be processed. Together, these fields tell the receiver what to do. Specific commands are presented later in this document.

Reply Packet Header Fields

error code

This field is used to indicate whether the command packet that is being replied to was successfully processed. A value of zero indicates success. A non-zero value indicates an error. The error code returned may be specific to each command set/command.

Data

The data field is unique to each command set/command. It is also different between command and reply packet pairs. For example, a command packet that requests a field value contains references to the object and field ids for the desired value in its data field. The reply packet's data field contains the value of the field.

Detailed Command Information

In general, the data field of a command or reply packet is an abstraction of a group of multiple fields that define the command or reply data. Each subfield of a data field is encoded in big endian format (See the *Java™ Virtual Machine Specification* for the definition of big-endian.) The detailed composition of data fields for each command and its reply are described in this section.

There is a small set of common data types that are common to many of the different KDWP commands and replies. They are described below.

Name	**Size**	**Description**
`Byte`	1 byte	A byte value.
`Boolean`	1 byte	A boolean value. `TRUE` is encoded as a non-zero value.
`Int`	4 bytes	A four-byte signed integer value.
`Long`	8 bytes	An eight-byte signed integer value.
`ObjectID`	4 bytes	Uniquely identifies an object in the target VM. A particular object is identified by exactly one objectID in KDWP commands and replies throughout its lifetime. An objectID of 0 represents a null object.
`ThreadID`	4 bytes	Uniquely identifies thread objects in the KVM.
`referenceTypeID`	4 bytes	Uniquely identifies a reference type in the target VM. It should not be assumed that for a particular class, the class `ObjectID` and the `referenceTypeID` are the same. Class, interfaces, and primitive data types are identified by `referenceTypeID`s. Each reference type has exactly one `referenceTypeID` during its lifetime.
`ArrayID`	4 bytes	Uniquely identifies references to arrays.
`MethodID`	4 bytes	Uniquely identifies a method in some class in the KVM. The `methodID`s for each method in a class must be unique. Since each `methodID` is paired with a `referenceTypeID` (which identifies the class or interface), `methodID`s do not need to be globally unique.
`FieldID`	8 bytes	Uniquely identifies a field in some class in the KVM. The `fieldID`s must be globally unique since referencing a field in an object may require the KVM to access field offsets in superclasses of the current object. The upper 4 bytes are the `classID` of the class that defines this field. The lower 4 bytes identify the field in this class.
`FrameID`	4 bytes	Uniquely identifies a frame in the KVM. The `frameID` must uniquely identify the frame within the entire KVM; it must be unique across all threads.

Location	13 bytes	An executable location. The location is identified by one byte type tag followed by a a `referenceTypeID` followed by a `methodID` followed by an unsigned eight-byte index, which identifies the location within the method. Index values are restricted as follows: The index of the start location for the method is less than all other locations in the method. The index of the end location for the method is greater than all other locations in the method. Index values within a method are monotonically increasing from the first executable point in the method to the last. For many implementations, each byte-code instruction in the method has its own index, but this is not required. The `type` tag is necessary to identify whether location's `referenceTypeID` identifies a class or an interface. Almost all locations are within classes, but it is possible to have executable code in the static initializer of an interface.
Value	Variable	A value retrieved from the target VM. The first byte is a signature byte which is used to identify the type. See `KDWP.Tag` for the possible values of this byte.Value's length is variable `byte`: 1-byte `short` or `char`: 2-bytes `int`: 4-bytes `long`: 8-bytes
Untagged value	Variable	A value as described above without the signature byte. This form is used when the signature information can be determined from context.
`String`	Variable	A UTF-8 encoded string, not zero terminated, preceded by a four-byte integer length.

1.4 Protocol details

Note – The commands in each command set are numbered to match the equivalent JDWP (Java Debug Wire Protocol) commands. Commands that are missing from the following list are presumed to be handled via the Debug Agent or are not necessary for minimal debugger functionality. Implementers could extend the KVM command set to handle more of the JDWP commands.

`VirtualMachine` *command set (1)*

`AllClasses` (3)
`AllThreads` (4)
`Suspend` (8)
`Resume` (9)
`Exit` (10)

`ReferenceType` *command set (2)*

`GetValues` (6)

`ClassType` *command set (3)*

`Superclass` (1)
`SetValues` (2)

`ObjectReference` *command set (9)*

`ReferenceType` (1)
`GetValues` (2)
`SetValues` (3)

`StringReference` *command set (10)*

`Value` (1)

`ThreadReference` *command set (11)*

`Name` (1)
`Suspend` (2)
`Resume` (3)
`Status` (4)
`Frames` (6)
`FrameCount` (7)
`Stop` (10)
`SuspendCount` (12)

`ArrayReference` *command set (13)*

`Length` (1)
`GetValues` (2)
`SetValues` (3)

`EventRequest` *command set (15)*

`Set` (1)

Event types:

```
Class prepare
Breakpoint
Midlet death
```

`Clear` (2)

`ClearAllBreakpoints` (3)

`StackFrame` *command set (16)*

`GetValues` (1)
`SetValues` (2)

`Event` *Command Set (64)*

`Composite` (100)

`KVM Vendor Specific` *Command set (128)*

`Handshake` (1)

Sends handshake string to KVM.
returns a 32 bit value that describes the capabilities of the KVM.

2

`VirtualMachine` Command Set

This command set is numbered (1) to match the equivalent JDWP command set.

`AllClasses` *Command (3)*

Returns reference types for all classes currently loaded by the target VM.

Out Data

(None)

Reply Data

int	*classes*	Number of reference types that follow.
Repeated *classes* times:		
`byte`	*refTypeTag*	Kind of following reference type.
`referenceTypeID`	*typeID*	Loaded reference type
`string`	*signature*	The JNI signature of the loaded reference type
`int`	*status*	The current class status.

`AllThreads` *Command (4)*

Returns all threads currently running in the target VM. The returned list contains threads created through `java.lang.Thread`. Threads that have not yet been started and threads that have completed their execution are not included in the returned list.

Out Data

(None)

Reply Data

`int`	*threads*	Number of threads that follow.
Repeated *threads* times:		
`threadID`	*thread*	A running thread

`Suspend` *Command (8)*

Suspends the execution of the application running in the target VM. All Java threads currently running are suspended.

Unlike `java.lang.Thread.suspend`, suspends of both the virtual machine and individual threads are counted. Before a thread can run again, it must be resumed through the VM-level suspend command or the thread-level suspend command the same number of times it has been suspended.

Out Data

(None)

Reply Data

(None)

`Resume` *Command (9)*

Resumes execution of the application after the suspend command or an event has stopped it. Suspensions of the Virtual Machine and individual threads are counted. If a particular thread is suspended `n` times, it must be resumed `n` times before it can continue.

Out Data

(None)

Reply Data

(None)

`Exit` ***Command (10)***

Terminates the target VM with the given exit code. All ids previously returned from the target VM become invalid. Threads running in the VM are abruptly terminated. A thread death exception is not thrown and finally blocks are not run.

Out Data

`int`	*exitCode*	The exit code

Reply Data

(None)

3

`ReferenceType` Command Set

This command set is numbered (2) to match the equivalent JDWP command set.

`GetValues` *Command (6)*

Returns the value of one or more static fields of the reference type. Each field must be a member of the reference type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced. For example, the values of private fields can be obtained.

Out Data

`referenceTypeID`	*refType*	The reference type ID.
`int`	*fields*	The number of values to get
Repeated *fields* times:		
`fieldID`	*fieldID*	A field to get

Reply Data

`int`	*values*	The number of values returned
Repeated *values* times:		
`value`	*value*	The field value

4

`ClassType` Command Set

This command set is numbered (3) to match the equivalent JDWP command set.

`Superclass` *Command (1)*

Returns the immediate superclass of a class.

Out Data

`classID`	*clazz*	The class type ID.

Reply Data

`classID`	*superclass*	The *superclass* (`NULL` if the class ID for `java.lang.Object` is specified).

`SetValues` *Command (2)*

Sets the value of one or more static fields. Each field must be a member of the class type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced. For example, the values of private fields can be set. Final fields cannot be set. For primitive values, the value's type must match the field's type exactly. For object values, there must exist a widening reference conversion from the value's type to the field's type and the field's type must be loaded.

Out Data

`classID`	*clazz*	The class type ID.
`int`	*values*	The number of fields to set.
Repeated *values* times:		
`fieldID`	*fieldID*	Field to set.
untagged-value	*value*	Value to put in the field.

Reply Data

(None)

5

`ObjectReference` Command Set

This command set is numbered (9) to match the equivalent JDWP command set.

`ReferenceType` *Command (1)*

Returns the runtime type of the object. The runtime type is a class or an array.

Out Data

`objectID`	*object*	The object ID

Reply Data

`byte`	*refTypeTag*	Kind of following reference type.
`referenceTypeID`	*typeID*	The runtime reference type.

`GetValues` *Command (2)*

Returns the value of one or more instance fields. Each field must be a member of the object's type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced. For example, the values of private fields can be obtained.

Out Data

`objectID`	*object*	The object ID
`int`	*fields*	The number of values to get
Repeated *fields* times:		
`fieldID`	*fieldID*	Field to get.

Reply Data

`int`	*values*	The number of values returned
Repeated *values* times:		
`value`	*value*	The field value

`SetValues` *Command (3)*

Sets the value of one or more instance fields. Each field must be a member of the object's type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, the values of private fields can be set. For primitive values, the value's type must match the field's type exactly. For object values, there must be a widening reference conversion from the value's type to the field's type and the field's type must be loaded.

Out Data

`objectID`	*object*	The object ID
`int`	*values*	The number of fields to set.
Repeated *values* times:		
`fieldID`	*fieldID*	Field to set.
`untagged-value`	*value*	Value to put in the field.

Reply Data

(None)

6

`StringReference` Command Set

This command set is numbered (10) to match the equivalent JDWP command set.

`Value` *Command (1)*

Returns the characters contained in the string.

Out Data

`objectID`	*stringObject*	The String object ID.

Reply Data

`string`	*stringValue*	The value of the String.

7

`ThreadReference` Command Set

This command set is numbered (11) to match the equivalent JDWP command set.

`Name` *Command (1)*

Returns the thread name.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

`string`	*threadName*	The thread name.

`Suspend` *Command (2)*

Suspends the thread.

Unlike `java.lang.Thread.suspend()`, suspends of both the virtual machine and individual threads are counted. Before a thread can run again, it must be resumed the same number of times it has been suspended.

Suspending single threads with this command has the same dangers as is the case with method `java.lang.Thread.suspend()`. If the suspended thread holds a monitor needed by another running thread, deadlock is possible in the target VM (at least until the suspended thread is resumed again).

The suspended thread is guaranteed to remain suspended until resumed through one of the JDI resume methods mentioned above.

Note that this doesn't change the status of the thread (see the `ThreadStatus` command.) For example, if it was `Running`, it still appears to other threads to be running.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

(None)

`Resume` ***Command (3)***

Resumes the execution of a given thread. If this thread was not previously suspended by the front-end, calling this command has no effect. Otherwise, the count of pending suspends on this thread is decremented. If it is decremented to 0, the thread continues to execute.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

(None)

`Status` ***Command (4)***

Returns the current status of a thread. The thread status reply indicates the thread status the last time it was running. The suspend status provides information on the thread's suspension, if any.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

`int`	*threadStatus*	One of the thread status codes. See `KDWP.ThreadStatus`
`int`	*suspendStatus*	One of the suspend status codes. See `KDWP.SuspendStatus`

`Frames` *Command (6)*

Returns the current call stack of a suspended thread. The sequence of frames starts with the currently executing frame, followed by its caller, and so on. The thread must be suspended, and the returned `frameID` is valid only while the thread is suspended.

Out Data

`threadID`	*threadObject*	The thread object ID.
`int`	*startFrame*	The index of the first frame to retrieve.
`int`	*length*	The count of frames to retrieve (-1 means all remaining).

Reply Data

`int`	*frames*	number of frames retrieved
Repeated *frames* times:		
`frameID`	*frameID*	The ID of this frame.
`location`	*location*	The current location of this frame

`FrameCount` *Command (7)*

Returns the count of frames on this thread's stack. The thread must be suspended, and the returned count is valid only while the thread is suspended.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

`int`	*frameCount*	The count of frames on this thread's stack.

`Stop` *Command (10)*

Stops the thread with an asynchronous exception, as if done by `java.lang.Thread.stop`.

Out Data

`threadID`	*threadObject*	The thread object ID.
`objectID`	*throwable*	Asynchronous exception. This object must be an instance of `java.lang.Throwable` or a subclass

Reply Data

(None)

`SuspendCount` *Command (12)*

Get the suspend count for this thread. The suspend count is the number of times the thread has been suspended through the thread-level or VM-level suspend commands without a corresponding resume.

Out Data

`threadID`	*threadObject*	The thread object ID.

Reply Data

`int`	*suspendCount*	The number of outstanding suspends of this thread.

8

`ArrayReference` Command Set

This command set is numbered (13) to match the equivalent JDWP command set.

`Length` *Command (1)*

Returns the number of components in a given array.

Out Data

`arrayID`	*arrayObject*	The array object ID.

Reply Data

`int`	*arrayLength*	The length of the array.

`GetValues` *Command (2)*

Returns a range of array components. The specified range must be within the bounds of the array.

Out Data

`arrayID`	*arrayObject*	The array object ID.
`int`	*firstIndex*	The first index to retrieve.
`int`	*length*	The number of components to retrieve. If length == -1, retrieve all components of the array.

Reply Data

`byte`	*Type tag*	The type of the components of the array
`Int`	*Length*	The number of components being returned.
`Values`	*Values*	Type tagged KDWP. Tag values of each component being returned.

`SetValues` *Command (3)*

Sets a range of array components. The specified range must be within the bounds of the array. For primitive values, each value's type must match the array component type exactly. For object values, there must be a widening reference conversion from the value's type to the array component type and the array component type must be loaded.

Out Data

`arrayID`	*arrayObject*	The array object ID.
`int`	*firstIndex*	The first index to set.
`int`	*values*	The number of values to set.
Repeated *values* times:		
`untagged-value`	*value*	A value to set.

Reply Data

(None)

9

`EventRequest` Command Set

This command set is numbered (15) to match the equivalent JDWP command set.

`Set` *Command (1)*

Set an event request. When the event described by this request occurs, an `event` is sent from the target VM.

Out Data

`byte`	*eventKind*	Event kind to request. See `KDWP.EventKind` for a complete list of events that can be requested. The default is to support only `Breakpoint`, `Class_Prepare` and `Midlet_Death` events.
`byte`	*suspendPolicy*	What threads are suspended when this event occurs? Note that the order of events and command replies accurately reflects the order in which threads are suspended and resumed. For example, if a `VM-wide resume` is processed before an event occurs which suspends the VM, the reply to the `resume` command is written to the transport before the suspending event. Refer to SuspendPolicy Constants in the Appendix.

`int`	*modifiers*	Constraints used to control the number of generated events. *Modifiers* specify additional tests that an event must satisfy before it is placed in the event queue. Events are filtered by applying each modifier to an event in the order they are specified in this collection Only events that satisfy all modifiers are reported. Filtering can improve debugger performance dramatically by reducing the amount of event traffic sent from the target VM to the debugger VM.
Repeated *modifiers* times:		
`byte`	*modKind*	Modifier kind
Case `ClassOnly` - if *modKind* is 4:		For `class prepare` events, restricts the events generated by this request to be the preparation of the given reference type and any subtypes. For other events, restricts the events generated by this request to those whose location is in the given reference type or any of its subtypes. An event is generated for any location in a reference type that can be safely cast to the given reference type. This modifier can be used with any event kind except `class unload`, `thread start`, and `thread end`.
`referenceTypeID`	*clazz*	Required class
Case `LocationOnly` - if *modKind* is 7:		Restricts reported events to those that occur at the given location. This modifier can be used with `breakpoint`, `field access`, `field modification`, `step`, and `exception` event kinds.
`location`	*loc*	Required location

Reply Data

`int`	*requestID*	ID of created request

`Clear` *Command (2)*

Clear an event request.

Out Data

`byte`	*event*	Event type to clear
`int`	*requestID*	ID of request to clear

Reply Data

(None)

`ClearAllBreakpoints` *Command (3)*

Remove all set breakpoints.

If bit 14 in the handshake is set, which means that the KVM stores event information, then the `ClearAllBreakpoints` command should be supported. Otherwise it is not supported.

Out Data

(None)

Reply Data

(None)

10

`StackFrame` Command Set

This command set is numbered (16) to match the equivalent JDWP command set.

`GetValues` *Command (1)*

Returns the value of one or more local variables in a given frame. Each variable must be visible at the current frame code index. Even if local variable information is not available, values can be retrieved if the front-end is able to determine the correct local variable index. (Typically, this index can be determined for method arguments from the method signature without access to the local variable table information.)

Out Data

`threadID`	*threadObject*	The frame's thread.
`frameID`	*frame*	The frame ID.
`int`	*slots*	The number of values to get.
Repeated *slots* times:		
`int`	*slot*	The local variable's index in the frame.
`byte`	*sigbyte*	A tag identifying the type of the variable

Reply Data

`int`	*values*	The number of values retrieved.
Repeated *values* times:		
`value`	*slotValue*	The value of the local variable.

`SetValues` *Command (2)*

Sets the value of one or more local variables. Each variable must be visible at the current frame code index. For primitive values, the value's type must match the variable's type exactly. For object values, there must be a widening reference conversion from the value's type to the variable's type and the variable's type must be loaded.

Even if local variable information is not available, values can be set, if the front-end is able to determine the correct local variable index. (Typically, this index can be determined for method arguments from the method signature without access to the local variable table information.)

Out Data

`threadID`	*threadObject*	The frame's thread.
`frameID`	*frame*	The frame ID.
`int`	*slotValues*	The number of values to set.
Repeated *slotValues* times:		
`int`	*slot*	The slot ID.
`value`	*slotValue*	The value to set.

Reply Data

(None)

11

`Event` Command Set

This command set is numbered (64) to match the equivalent JDWP command set. Note that by default, KDWP supports only `Breakpoint`, `Class_Prepare` and `Midlet_Death` events.

`Composite` *Command (100)*

Several events may occur at a given time in the target VM. For example, there might be more than one breakpoint request for a given location, or you might single step to the same location as a breakpoint request. These events are delivered together as a composite event. For uniformity, a composite event is always used to deliver events, even if there is only one event to report.

The events that are grouped in a composite event are restricted in the following ways:

- Only with other `class prepare` events for the same class:
  - `Class Prepare` Event
- Only with other members of this group, at the same location and in the same thread:
  - `Breakpoint` Event

Event Data

`byte`	*suspendPolicy*	Which threads were suspended by this composite event?
`int`	*events*	Events in set.
Repeated *events* times:		
`byte`	*eventKind*	Event kind selector

Case `Breakpoint` - if *eventKind* is `KDWP.EventKind.BREAKPOINT`:		Notification of a breakpoint in the target VM. The breakpoint event is generated before the code at its location is executed.
`int`	*requestID*	Request that generated event
`threadID`	*thread*	Thread that hit breakpoint
`location`	*location*	Location hit
Case `ClassPrepare` - if *eventKind* is `KDWP.EventKind.CLASS_PREPARE`:		Notification of a class prepare in the target VM. See the *Java™ Virtual Machine Specification* for a definition of class preparation. Class prepare events are not generated for primitive classes (for example, `java.lang.Integer.TYPE`).
`int`	*requestID*	Request that generated event
`threadID`	*thread*	Preparing thread. In rare cases, this event might occur in a debugger system thread within the target VM. Debugger threads take precautions to prevent these events, but they cannot be avoided under some conditions, especially for some subclasses of `java.lang.Error`. If the event was generated by a debugger system thread, the value returned by this method is `NULL`, and if the requested suspend policy for the event was `EVENT_THREAD` all threads are suspended instead, and the composite event's suspend policy reflects this change. Note that this does not apply to system threads created by the target VM during its normal (non-debug) operation.
`byte`	*refTypeTag*	Kind of reference type. See `KDWP.TypeTag`
`referenceTypeID`	*typeID*	Type being prepared
`string`	*signature*	Type signature
`int`	*status*	Status of type. See `KDWP.ClassStatus`
Case `Midlet` Death – if *`eventKind`* is `KDWP.EventKind.MIDLET_DEATH`		Notification of a completed midlet in the target VM. The notification is generated by the dying midlet before it terminates.
`Int`	*RequestID*	Request that generated event.
`String`	*MidletName*	JNI signature of the dying Midlet.

Reply Data

For `Breakpoint` type events, returns the byte opcode that was originally in the location that currently has the breakpoint.

`Byte`	*Opcode*	Original opcode that was at the breakpoint.

12

Vendor Specific Command Set

This command set is numbered (128) to match the equivalent JDWP command set.

Handshake *Command (1)*

Used to initialize communication between the Debug Agent (DA) and the KVM. The KVM determines if the DA is the correct one for this particular KVM. If so, then the KVM replies with a 32-bit bitfield that indicates any optional JDWP commands that the KVM is able to parse directly (meaning that the DA can pass these JDWP commands directly to the KVM without parsing/managing them). Whether out data or reply, if the ID string is the `NULL` string (its length is 0) then the receiver of the ID string ignores it.

Out Data

`String`	Identifier	Vendor specific ID string
`byte`	Major Version	The major version of the Debug Agent.
`Byte`	Minor Version	The minor version of the Debug Agent.

Reply Data

`String`	*Identifier*	Vendor specific ID string.
int	*Optional commands*	32-bit bitfield that describes the optional JDWP commands that the KVM supports. This set of bits is in 'Network Order' (Big Endian) format.
Bit 0	*VM Init event*	KVM supports/sends `VM_INIT` event.

Bit 1	*VM Death*	KVM supports/sends `VM_DEATH` event
Bit 2	*Method Entry Event*	KVM supports/sends `METHOD_ENTRY` event
Bit 3	*Method Exit Event*	KVM supports/sends `METHOD_EXIT` event
Bit 4	*Exception Event*	KVM supports/sends `EXCEPTION` event
Bit 5	*Exception Catch Event*	KVM supports/sends `EXCEPTION_CATCH` event
Bit 6	*Class Load Event*	KVM supports/sends `CLASS_LOAD` event
Bit 7	*Class unload Event*	KVM supports/sends `CLASS_UNLOAD` event
Bit 8	*Single Step Event*	KVM supports/sends `SINGLE_STEP` event
Bit 9	*Thread start Event*	KVM supports/sends `THREAD_START` event
Bit 10	*Thread death Event*	KVM supports/sends `THREAD_DEATH` event
Bit 11	*Frame pop Event*	KVM supports/sends `FRAME_POP` event
Bit 12	*Field Access Event*	KVM supports/sends `FIELD_ACCESS` event
Bit 13	*Field modification Event*	KVM supports/sends `FIELD_MODIFICATION` event
Bit 14	*Event management*	If set then the KVM keeps a list of events that have been set by the debugger and does not need the debug agent to return the breakpoint opcode after a breakpoint event.

13

Appendix: Constants

`ClassStatus` *Constants*

`PREPARED`	2	
`VERIFIED`	1	
`INITIALIZED`	4	
`ERROR`	8	

`ThreadStatus` *Constants*

`RUNNING`	1	
`WAIT`	4	
`SLEEPING`	2	
`ZOMBIE`	0	
`MONITOR`	3	

`TypeTag` *Constants*

`CLASS`	1	`ReferenceType` is a class.
`INTERFACE`	2	`ReferenceType` is an interface.
`ARRAY`	3	`ReferenceType` is an array.

`Tag` *Constants*

`ARRAY`	91	'[' - an array object (`objectID` size).
`BYTE`	66	'B' - a byte value (1 byte).
`CHAR`	67	'C' - a character value (2 bytes).
`OBJECT`	76	'L' - an object (`objectID` size).
`FLOAT`	70	'F' - a float value (4 bytes).
`DOUBLE`	68	'D' - a double value (8 bytes).
`INT`	73	'I' - an int value (4 bytes).
`LONG`	74	'J' - a long value (8 bytes).
`SHORT`	83	'S' - a short value (2 bytes).
`VOID`	86	'V' - a void value (no bytes).
`BOOLEAN`	90	'Z' - a boolean value (1 byte).
`STRING`	115	's' - a String object (`objectID` size).
`THREAD`	116	't' - a Thread object (`objectID` size).
`THREAD_GROUP`	103	'g' - a `ThreadGroup` object (`objectID` size).
`CLASS_LOADER`	108	'l' - a `ClassLoader` object (`objectID` size).
`CLASS_OBJECT`	99	'c' - a class object object (`objectID` size).

`Error` *Constants*

`INVALID_TAG`	500	object type id or class tag
`ALREADY_INVOKING`	502	previous invoke not complete
`INVALID_INDEX`	503	
`INVALID_LENGTH`	504	
`INVALID_STRING`	506	
`INVALID_CLASS_LOADER`	507	
`INVALID_ARRAY`	508	
`TRANSPORT_LOAD`	509	
`TRANSPORT_INIT`	510	

NATIVE_METHOD	511	
INVALID_COUNT	512	
VM_DEAD	112	
INVALID_MONITOR	50	
OUT_OF_MEMORY	110	
INVALID_SLOT	35	
INVALID_CLASS_FORMAT	60	
INVALID_THREAD	10	
INTERRUPT	52	
NOT_MONITOR_OWNER	51	
CIRCULAR_CLASS_DEFINITION	61	
ACCESS_DENIED	111	
INVALID_FIELDID	25	
TYPE_MISMATCH	34	
OPAQUE_FRAME	32	
CLASS_NOT_PREPARED	22	
FAILS_VERIFICATION	62	
INVALID_METHODID	23	
INVALID_CLASS	21	
INVALID_OBJECT	20	
ADD_METHOD_NOT_IMPLEMENTED	63	
NULL_POINTER	100	
DUPLICATE	40	
INVALID_FRAMEID	30	
UNATTACHED_THREAD	115	
THREAD_NOT_SUSPENDED	13	
INVALID_LOCATION	24	
INVALID_TYPESTATE	65	
THREAD_SUSPENDED	14	

ABSENT_INFORMATION	101	
INVALID_THREAD_GROUP	11	
INTERNAL	113	
NONE	0	
INVALID_PRIORITY	12	
ILLEGAL_ARGUMENT	103	
SCHEMA_CHANGE_NOT_IMPLEMENTED	64	
INVALID_EVENT_TYPE	102	
NOT_CURRENT_FRAME	33	
NOT_IMPLEMENTED	99	
NO_MORE_FRAMES	31	
NOT_FOUND	41	

EventKind Constants

VM_START	90	
THREAD_DEATH	7	
METHOD_EXIT	41	
EXCEPTION_CATCH	30	
USER_DEFINED	5	
METHOD_ENTRY	40	
VM_DEATH	99	
CLASS_UNLOAD	9	
CLASS_PREPARE	8	
SINGLE_STEP	1	
FIELD_MODIFICATION	21	
CLASS_LOAD	10	
THREAD_START	6	
FRAME_POP	3	
VM_INIT	90	
BREAKPOINT	2	
THREAD_END	7	
FIELD_ACCESS	20	
EXCEPTION	4	
MIDLET_DEATH	100	

SuspendStatus Constants

SUSPEND_STATUS_SUSPENDED	0x1	

`SuspendPolicy` *Constants*

`NONE`	0	Suspend no threads when this event is encountered.
`EVENT_THREAD`	1	Suspend the event thread when this event is encountered.
`ALL`	2	Suspend all threads when this event is encountered.