MAC OS RUNTIME FOR JAVA

Programming Note

Using JDirect to Access Mac OS Code From Java

For MRJ 2.0

PRELIMINARY 12/17/97 Technical Publications © Apple Computer, Inc. 1997 Apple Computer, Inc.
 © 1997 Apple Computer, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except to make a backup copy of any documentation provided on CD-ROM.

The Apple logo is a trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers. Every effort has been made to ensure that the information in this

manual is accurate. Apple is not responsible for typographical errors. Apple Computer, Inc.

1 Infinite Loop Cupertino, CA 95014 408-996-1010

Apple, the Apple logo, Mac, MacinTalk, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions. Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Calling Mac OS Code Using JDirect5Parameter Passing between Java and C7Using Wrapper Objects9Accessing Memory9Representing Strings12Representing C Data Structures12

JDirect is a programming interface that allows you to access Mac OS C code from a Java[™] application. You can use JDirect if you have either of the following needs:

- You want to access Mac OS system functions from your Java application
- You want your Java application to use older legacy code written for the Mac OS platform

For example, to be a good Mac OS citizen, an application must have a bundle (creator and type registry) so the Finder will know which application to launch when a document is double clicked. You can add a 'BNDL' resource and reserve a unique creator code for your application (and attach them using JBindery), but if you are writing your application in Java, you cannot ensure that documents your application creates will have the correct creator and type.

Note

You can access many basic Mac OS Toolbox functions using MRJToolkit, which uses JDirect to handle calls. Before using JDirect, you should make sure that MRJToolkit does not already handle the functionality that you need. ◆

You can use JDirect with any Java application that can run in the Mac OS Runtime for Java (MRJ) Java environment.

Calling Mac OS Code Using JDirect

Traditionally, if you wanted to access Mac OS methods from Java code, you had to take the following steps:

- 1. Create a Java class that defines your Mac OS function.
- 2. Use the javah tool on the class file to create a C stub function that calls the Mac OS function.
- 3. Create a shared library that exports the stub function.
- 4. Call the System.LoadLibrary method from your Java application to load the library before making the call to the Mac OS function.

Using JDirect to Access Mac OS Code From Java

In addition to these steps, you would have the overhead of having to maintain separate Java and C projects, and you would have to run javah and build a new stub library every time you added a new method.

JDirect lets you access Mac OS code with minimal overhead and no extra C code. To use JDirect to access Mac OS code, you must take the following steps:

- 1. Create a class that implements com.apple.NativeObject.
- 2. Declare your Mac OS methods within the class. These methods must be static, and they must have the same case-sensitive name as the corresponding symbol exported from the Mac OS shared library fragment.
- 3. Define a static String array named kNativeLibraryNames within the class. This array should contain the names of the shared library fragments that export the desired symbols.

IMPORTANT

The shared library fragment name is not necessarily the same as the name of the shared library file (for example, several different shared library fragments may be packaged in one shared library file). For more information about fragments, see *Mac OS Runtime Architectures*. ▲

Listing 1-1 shows an example of using JDirect to call the Mac OS Toolbox function SysBeep contained in the fragment InterfaceLib.

Listing 1-1 Calling SysBeep from Java code

```
import com.apple.NativeObject;
public class Beeper implements NativeObject {
    public native static void SysBeep(short duration);
    private static String[] kNativeLibraryNames = { "InterfaceLib" };
    }
```

When a class is loaded, MRJ checks to see if the class implements a special interface, com.apple.NativeObject. If it does, any Mac OS methods are linked from the shared libraries named by the String array, kNativeLibraryNames. This list of shared libraries is local to each class, so even if two different classes have

Using JDirect to Access Mac OS Code From Java

Mac OS methods of the same name, each class will get the method from its own private list of libraries.

In cases where you might not be sure which library exports a symbol, you can include all the possible candidates in the String array. JDirect then searches the libraries in the order they are listed.

Note

You can subclass any class that implements com.apple.NativeObject. ◆

Parameter Passing between Java and C

When the Mac OS method is called, the actual signature of the method (the types of its arguments) is used to build a parameter list for the Mac OS function. All of the basic Java types – byte, char, short, int, long, float, and double – are passed as is, by value, in the order specified. Table 1-1 shows the correspondence between Java and Mac OS C data types.

Java Type	С Туре
boolean	Boolean or unsigned char
byte	signed char
char	unsigned short
short	signed short
int	signed long
long	signed long long
float	float
double	double

Java versus C data types

IMPORTANT

Table 1-1

You should not use the Java long type (long long in C) as a return value. ▲

Using JDirect to Access Mac OS Code From Java

Array references are passed as a pointer to the first element of the array.

Table 1-2 shows some examples of C function declarations and their corresponding Java versions.

 Table 1-2
 Some C function declarations and their Java versions.

C Version	Java Version
<pre>extern short MyGetResCount(ResType theType);</pre>	<pre>static native short MyGetResCount(int theType);</pre>
extern Handle MyGetResource(ResType theType, short theID);	<pre>static native int MyGetResource(int theType, short theID);</pre>
<pre>extern long MyHashFunction (const char* cString);</pre>	<pre>static native int MyHashFunction(byte [] cString);</pre>
extern Boolean MyCompareString(ConstStr255Param s1, ConstStr255Param s2);	<pre>static native boolean MyCompareString(byte [] s1, byte [] s2);</pre>
extern void MyBlockCopy (const void* src, void* dst, unsigned long length);	<pre>static native void MyBlockCopy(byte [] src, byte [] dst, int length);</pre>

Any objects (such as data structures) must be encapsulated in a "wrapper," which hides the representation of the Mac OS object. If you choose not to use the Apple-defined wrapper classes described in "Using Wrapper Objects" (page 9) to encapsulate your objects, you must create your own. That is, you must define any objects as <code>byte[]</code> arrays and enforce a field alignment within it. Then, to access any fields (that is, elements in the array), you must define additional methods to read them (for example, <code>getByteAt(offset)</code>).

IMPORTANT

Your Mac OS function or method should not retain object and array pointer references, as these are likely to become invalid soon after the call. This is because Java garbage collectors are free to compact the heap and move the contents of objects around in memory. Some objects, such as windows, graphics ports, and anything represented as a handle, must be managed in a separate heap, using wrapper objects. ▲

Using Wrapper Objects

WARNING WARNING

The Apple-defined wrapper classes described in this section are subject to change and are *not* guaranteed to work with versions of MRJ later than 2.0 \blacktriangle

To access Mac OS functions, the Java code must often be able to do things it was not meant to do, such as manipulate pointers and handles. The way to accomplish this is through wrapper classes, which hide the representation of the Mac OS data structure. If you want to call your own Mac OS code, you will have to create wrapper classes that encapsulate your data types.

MRJ 2.0 provides primitive classes in the package com.apple.memory, which lets you manipulate memory from your Java code.

Accessing Memory

As mentioned earlier, the basic approach to representing toolbox objects in Java is to provide wrapper classes that hide the representation of the Mac OS data structure. Accessing the data inside such objects is accomplished by writing "get" and "set" methods. Since toolbox objects are either pointer-based or handle-based, we provide base-class wrapper objects, PointerObject, and HandleObject, which provide get/set methods for primitive values at a known offset. Both are subclasses of the abstract class MemoryObject, which is shown in Listing 1-2.

Listing 1-2 The MemoryObject Class

```
package com.apple.memory;
public abstract class MemoryObject implements NativeObject {
    protected abstract byte getByteAt(int offset);
    protected abstract short getShortAt(int offset);
    protected abstract int getIntAt(int offset);
    protected abstract long getLongAt(int offset);
```

```
CHAPTER 1
```

```
protected abstract float getFloatAt(int offset);
protected abstract double getDoubleAt(int offset);
protected abstract void setByteAt(int offset, byte value);
protected abstract void setShortAt(int offset, short value);
protected abstract void setIntAt(int offset, int value);
protected abstract void setLongAt(int offset, long value);
protected abstract void setFloatAt(int offset, float value);
protected abstract void setDoubleAt(int offset, double value);
protected abstract int getSize();
protected abstract int getSize();
}
```

PointerObject and HandleObject each implement these abstract methods using either pointer or handle dereferencing. Listing 1-3 shows the PointerObject class, and Listing 1-4 shows the HandleObject class.

Listing 1-3 The PointerObject class

```
package com.apple.memory;
public class PointerObject extends MemoryObject {
    /**
    * Creates a PointerObject from an existing pointer
    * @param address the address of an existing pointer
    * @param size the size of the memory block pointed to
    */
    public PointerObject(int address, int size) { ... }
    public final int getPointer() { return this.pointer; }
    protected int pointer;
    ...
```

10

```
CHAPTER 1
```

Listing 1-4 The HandleObject class

```
package com.apple.memory;
public class HandleObject extends MemoryObject {
  /**
 * Creates a HandleObject from the specified memory handle.
 * @param handle a handle to memory.
 */
    public HandleObject(int handle) { ... }
    public final int getHandle() { return this.handle; }
    protected int handle;
    ...
}
```

Subclasses of PointerObject or HandleObject can then define higher-level get/ set methods to present a familiar interface to the Mac OS data structures. Listing 1-5 shows an example wrapper class for a MacOS window record.

Listing 1-5 A wrapper class for a Window Record

```
class WindowRef extends PointerObject {
   public boolean isVisible() {
      return (getByteAt(110) != 0); // offsetof(WindowRecord, visible)
      }
}
```

This simple example shows how Mac OS Boolean values are mapped to Java boolean values. Since any nonzero value can be a true value in C , we represent Mac OS Boolean values as byte values in Java, which we convert to a type boolean by comparing with zero.

The value 110 is the offset of the visible field in the WindowRecord data structure.

Using JDirect to Access Mac OS Code From Java

Representing Strings

Since JDirect allows you to pass a Java array directly to Mac OS functions as a pointer to its first array element, you can represent C and Pascal strings as byte[] array objects. C strings are merely byte arrays with an extra zero byte to signal the end of string. Pascal strings are simply byte arrays with the first element indicating the length of the string. Note that since some Mac OS functions accept C strings (such as those in the standard C library) and others require Pascal strings, JDirect does not handle strings automatically.

Representing C Data Structures

Not all Mac OS objects are easily represented in Java. For example, C structures can contain embedded arrays, as well as nested structures. In Java, arrays are always separate objects, and fields that refer to objects are always references. You can solve this representation problem by using an opaque representation of the object defined by the ByteObject class.

Consider the file system specification record (FSSpec) in Listing 1-6.

Listing 1-6 A file system specification record

```
struct FSSpec {
    short vRefNum;
    long parID;
    unsigned char name[64];
    };
```

You can think of the structure as being one long array of byte values. You can then access a field using offsets. For example, the parID field would begin at an offset of 2 from the beginning of the structure (since vRefNum takes up 2 bytes).

The ByteObject class is a subclass of MemoryObject that provides a way to represent a structure as a byte[] array. Listing 1-7 shows the ByteObject class.

Listing 1-7 The ByteObject class

```
package com.apple.memory;
public class ByteObject extends MemoryObject {
/**
* Constructor that sets the size of the byte array.
* @param size the number of bytes for memory allocation.
* @return this.
*/
    protected ByteObject(int size) {...}
    /**
    * Returns the byte array containing the toolbox data structure.
    * @return the actual byte array NOT a copy
    */
   public final byte[] getBytes() { return bytes; }
   public final int getSize() { return bytes.length; }
   protected byte[] bytes;
    . . .
}
```

To access the fields in your structure, you must write get/set methods in your class. Listing 1-8 shows how you could implement the FSSpec structure by representing it as a byte[] array.

Listing 1-8 Accessing a file system specification record using the ByteObject class

```
class FSSpec extends ByteObject {
   static final int sizeOfFSSpec = 70; // 70 bytes in size
   FSSpec() {
      super(sizeOfFSSpec); // allocate 70 bytes please.
```

```
}
short getVRefNum() { return getShortAt(0); }
int getParID() { return getIntAt(2); }
String getName() {
    int length = name0; // filename is a Pascal string
    char value[] = new char[length];
    int offset = 7;
    for (int i = 0; i < length; i++)
        value[i] = (char)getByteAt(offset++);
        return new String(value);
    }
}</pre>
```

The getName method builds the name of the file from an array of characters.

<More information to be added later>

Using JDirect to Access Mac OS Code From Java

Using Wrapper Objects 12/17/97 © Apple Computer, Inc. This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe[™] Illustrator and Adobe Photoshop.

Text type is Palatino[®] and display type is Helvetica[®]. Bullets are ITC Zapf Dingbats[®]. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER Jun Suzuki