The Java Native Code API

Frank Yellin

1 Summary

The Java Native Code API is intended for all programmers intending to write native code generators or other utilities that run inside the Java Virtual machine.

The Java Native Code API is intended to support several different styles of native code generators:

- **Ahead-of-time recompilers**. This some of compiler rewrites a java class file into a so called "fat" class file. This fat class file must give both the original byte-code definition of all of the methods, and can also give one or more alternative native-machine code definitions for some of the methods. The "fat" class file can easily accommodate native-machine code definitions for several machine architectures, simultaneously.
- **Ahead-of-time compiler**: This sort of compiler converts Java (or other language) source code into a "fat" class file. Again, the fat class file must give both byte-code definitions of all the methods, and can also give one or more alternative native-machine code definitions for some of the methods. Compilers can often produce better code than recompilers, since they can know the original "intent" of the programmer.
- **Just In Time Code Generators**: A JIT code generator is intended to run concurrently with the execution of the Java Virtual Machine. The JIT code generator should determine those methods that are called most often, and generate machine code for them on-the-fly. A JIT code generator can make use of specific hardware or coprocessors running in the current environment.

Although this API is intended for use with compilers and code generators generating native code, this API can easily be extended by vendors to be used for other purposes. This API provides convenient hooks so that vendor-written code can be executed within the JVM. However for convenience, we call any vendor-written code a *compiler* if it conforms to this Native Code API.

Note that in all cases, it is assumed that there is some sort of external "shared" library that is to be linked in to the Java Virtual Machine. For ahead-of-time compilers, this library must include routines to read in the compiled code and to patch it up. For just-in-time compilers, this library must be able to compile JVM byte codes on the fly.

It is assumed that readers of this document are familiar with the Java language and with the Java Virtual Machine. Additional information on the virtual machine can be found in *The Java Virtual Machine* by Tim Lindholm and Frank Yellin

2 The class java.lang.Compiler

The class `java.lang.Compiler` is the principal application interface to the native code API. This class is responsible for both determining if a compiler is available, and if so initializing it. In addition, this class contains several methods by which the application can pass information to the compiler.

By default, all the methods in this class effectively do nothing. However as part of the compiler initialization step above, these methods can be modified to actually do useful work

Here are the methods in `java.lang.Compiler` and their intended meaning.

- `public static void disable().`

  Disable the compiler. The compiler is enabled by default. The implementation of this method is described more fully in §5.1.

- `public static void enable()`

  Enable the compiler if it has been disabled. The implementation of this method is described more fully in §5.1.

- `public static boolean compileClass(Class clazz)`

  Compile the indicated class. Return `true` if successful, `false` otherwise. The implementation of this method is described more fully in §5.2. The `clazz` argument is an instance of `java.lang.Class` representing a class in the running Java application.

- `public static boolean compileClasses(String string)`

  Compile all classes whose names have the indicated pattern. Return `true` if successful, `false` otherwise. The implementation of this method is described more fully in §5.2.

- `public static Object command(Object any)`

  This method has no predefined meaning.

3 Additional structures

Every method in the JVM has a runtime structure associated with it called the *method block*. This method block includes the following three fields:

```
void    *CompiledCode;
void    *CompiledCodeInfo;
long     CompiledCodeFlags;
```

These fields are never used by the Java Virtual Machine. Compiler implementors are free to use these fields however they wish.

4 Start-up

There are three actions that the JVM performs in order to initialize any compiler.

1. Before executing any Java code, set the C variable `UseLosslessQuickOpcodes` to `TRUE`.
2. Load in the class `java.lang.Compiler` using the normal class loading mechanism. This action causes this class's static initializer to execute. See §4.2 for more details.
3. If the compiler class could not be found, or if the compiler fails to initialize (see §4.2), the C variable `UseLosslessQuickOpcodes` is reset to `FALSE`.

4.1 UseLosslessQuickOpcodes

In the normal execution of the Java Virtual Machine, certain byte codes are rewritten into new, so-called "quick" byte codes that can be executed more efficiently. These quick byte codes are internal to the Java Virtual Machine and cannot normally appear in user code.

For example, Java code of the form

```
String x;
x.length();
```

is compiled to something like

0 invokevirtual #4 // *Method* `String.length()I`

where, for example, index 4 into the constant pool of the current class is a symbolic reference to the method `String.length`. However, after the first execution of this instruction, it is rewritten into

0 invokevirtual_quick 3 1

This rewrite indicates several things:

- that the `String` class does in fact have a `length()` method,
- the current method is entitled to call the `length()` method
- that in `String` or any subclass of `String`, the `length()` method must always be the third item in the class's method table.
- that `String.length()` has a total argument length of 1, including the "this" argument.

Future execution of this code can grab the "this" argument off the stack, find its method table, and get the third method in that method table. No other checks are necessary. This new instruction is much faster.

However the knowledge that the code is calling the specific method `String.length()` has been lost. Just in time compilers that want to perform code inlining or other such optimizations would be unable to determine what specific methods were intended to be called.

Setting the C variable `UseLosslessQuickOpcodes` to `TRUE` forces the Java Virtual machine to be very restrictive about the sorts of rewrites that it does. If this variable is `TRUE`, the JVM performs special rewrites for which no information is lost. In particular:

- All `opc_invokevirtual` byte codes are converted to `opc_invokevirtual_quick_w` byte codes. Normally, this instruction can turn into either `opc_invokevirtual_quick`, `opc_invokevirtualobject_quick`, or `opc_invokevirtual_quick_w`
- All `opc_getfield` are converted into `opc_getfield_quick_w` byte codes. Normally, this instruction can turn into either `opc_getfield_quick`, `opc_getfield2_quick`, or `opc_getfield_quick_w`.
- All `opc_putfield` are converted into `opc_putfield_quick_w` byte codes. Normally, this instruction can turn into either `opc_putfield_quick`, `opc_putfield2_quick`, or `opc_putfield_quick_w`.

4.2 Static initialize of java.lang.Compiler

Loading `java.lang.Compiler` causes the following static initializer to be executed:

```
try {
    String library =
            System.getProperty("java.compiler");
    if (library != null) {
        System.loadLibrary(library);
        initialize();
    }
} catch (Throwable e) { }
```

If the system property `java.compiler` is defined and is not `null`, it is treated as the name of a shared library. This shared library is loaded, and then the compiler is initialized through a call to the native method `initialize`.

The exact method in which system properties are defined and by which shared libraries are loaded is machine dependent.

The following is the native code that is executed by the call to the Java method `Compiler.initialize`.

```
*(void (*)(void **)) address = (void *)
      sysDynamicLink("java_lang_Compiler_start");
if (address != 0)
    address (CompiledCodeLinkVector)
compilerInitialized = TRUE;
```

1. We look for the address of the symbol `java_lang_Compiler_start` in the shared library. The exact method by which that address is determined is machine and implementation specific.
2. If that symbol is found, it is presumed to be the name of an initialization function. That function is called with `CompiledCodeLinkVector` (see §4.3, §5, §6) as its single argument.
3. The variable `compilerInitialized` is set to `TRUE`. If this variable maintains its initial value of `FALSE`, the JVM assumes that the compiler has not been initialized and sets `UseLosslessQuickOpcodes` (see §4.1) to `FALSE` on its own. If this variable is set to `TRUE`, then the compiler has been initialized, and its initialization function is responsible for setting `UseLosslessQuickOpcodes` to whatever value is appropriate.

4.3 CompiledCodeLinkVector

The sole argument to the compiler initialization function is a link vector. This link vector is an array of addresses of various functions and variables in the Java Virtual Machine.

On some operating systems, shared libraries can access global functions and variables that exist in the main program. However on many others, shared libraries and DLL's cannot access these global functions and variables. For this reason, the initialization routine above passes the value `CompiledCodeLinkVector` in Step 2. This link vector is an array of addresses in the main program that the compiler might need.

Each address is one of the following:

- The address of a hook (see §5) to the compiled code. The compiler initialization can modify these hooks to be the address of a function in the shared library.
- The address of a function that the compiler or compiled code might need to call.
- The address of variables whose value the compiler may want to change.
- The address of certain important constants, such as the class structure for `java.lang.String` or the array of all classes.

5 JVM Hooks in the Link Vector

There are several entries in the link vector that can be used by a compiler to affect the running of the Java Virtual Machine.

Each of these entries is the address of a variable. The initial value of each of these variables is initially 0. The compiler can set one or more of these variables to be a non-zero value in order to cause some compiler-specific function to be called at a specific time.

5.1 void (*p_CompilerEnable)(), void (*p_CompilerDisable)()

The two functions

```
java_lang_Compiler_enable()
java_lang_Compiler_disable()
```

are called whenever the application calls the Java methods `Compiler.enable`() and `Compiler.disable()` respectively

The definition of these two functions are as follows:

```
void java_lang_Compiler_enable(Hjava_lang_Compiler *this) {
    if (p_CompilerEnable != NULL)
        p_CompilerEnable();
}

void java_lang_Compiler_disable(Hjava_lang_Compiler *this) {
    if (p_CompilerDisable != NULL)
        p_CompilerDisable();
}
```

Compilers should set the values of the variables `p_CompilerDisable` and `p_CompilerEnable` to cause specific actions to occur when the methods `Compiler.disable` and `Compiler.enable` are invoked, respectively.

5.2 boolean (*p_CompilerCompileClass)(ClassClass *class) boolean (*p_CompilerCompileClasses)(Hjava_lang_String *name)

The function `java_lang_Compiler_CompilerCompileClass` is called whenever the application calls the Java method `Compiler.compileClass`.

The function `java_lang_Compiler_CompilerCompileClasses` is called whenever the application calls the Java method `Compiler.compileClasses.`

The definition of these functions is as follows:

```
long java_lang_Compiler_compileClass(
            Hjava_lang_Compiler *this,
            Hjava_lang_Class *clazz) {
    if (clazz == NULL) {
        SignalError(0, JAVAPKG "NullPointerException", 0);
        return FALSE;
    } else if (p_CompilerCompileClass != NULL) {
        return p_CompilerCompileClass(unhand(clazz));
    } else {
        return FALSE;
    }
}

long java_lang_Compiler_compileClasses(
        Hjava_lang_Compiler *this,
         Hjava_lang_String *name) {
    if (name == NULL) {
        SignalError(0, JAVAPKG "NullPointerException", 0);
        return FALSE;
    } else if (p_CompilercompileClasses != NULL) {
        return p_CompilercompileClasses(name);
    } else {
        return FALSE;
    }
}
```

Compilers should set the values of `p_CompilerCompileClass` and `p_CompilerCompileClasses` to cause more useful operations to occur when the Java methods are called.

The functions pointed at by the two variables should return a non-zero value if they have successfully compiled the class(es), and zero otherwise.

5.3 JHandle *(*p_CompilerCommand)(JHandle *)

The function

```
java_lang_Compiler_command()
```

is called whenever the application calls the Java method `Compiler.command`().

The definition of this function is as follows:

```
JHandle*
java_lang_Compiler_command(Hjava_lang_Compiler *this,
                           JHandle *x) {
    if (x == NULL) {
        SignalError(0, JAVAPKG "NullPointerException", 0);
        return NULL;
```

```
        } else if (p_CompilerCommand != NULL) {
            return p_CompilerCommand(x);
        } else {
            return NULL;
        }
    }
```

Compilers should set the values of the variables `p_CompilerCommand`* to cause specific actions to occur when the `Compiler.command` method is invoked. This method has no predefined meaning.

5.4 void (*p_InitializeForCompiler)(ClassClass *cb)

The function `InitializeForCompiler` is called during the final phase of a class's resolution. Every Java class is resolved before any method in it is called, any static variable in it is accessed, and before any instance of that class is created.

This function is called just after each method in the class has an "invoker" assigned to it, but before the class has been verified. Methods that already have native code attached to them (see §5.7), already have been assigned the invoker `InvokeCompiledMethod`. The definition of this function is as follows:

```
void InitializeForCompiler(ClassClass *cb) {
    if (p_InitializeForCompiler != NULL)
        p_InitializeForCompiler(cb);
}
```

The function pointed at by `p_InitializeForCompiler` can compile or recompile some of the methods, can change the invokers, or can modify the compiled code.

One important use of this function could be to "patch" precompiled code. For example, precompiled code might not know the exact offsets of specified fields or methods of other classes. Similarly, the precompiled code might contains calls to runtime functions whose address is not known at compile time.

5.5 (void *)(*p_CompilerFreeClass)(ClassClass *)

The function `CompilerFreeClass()` is called when a class is about to be removed from the Java Virtual Machine because there are no more references to it and no more instances of it. The definition of this function is as follows:

```
void CompilerFreeClass(ClassClass *cb) {
    if (p_CompilerFreeClass != NULL)
        p_CompilerFreeClass(cb);
}
```

The function pointed at by `p_CompilerFreeClass` should free any resources that the compiler has set aside for this class and perform any other necessary cleanup.

* The address of `java_lang_Compiler_command` seems to be missing from the link vector in the JDK 1.0.2. No one seems to have noticed its absence.

5.6 void (*p_CompiledCodeSignalHandler)(int sig, void *info, void *uc)

The function `CompiledCodeSignalHandler` is called by Java's interrupt handler[*] when the Java process receives an unexpected interrupt. The arguments are the identical to the arguments by which the signal handler is called. The variable `info` contains the siginfo structure, and the `uc` contains additional information. The definition of this function is as follows:

```
void CompiledCodeSignalHandler(int sig, void *info, void *uc) {
    if (p_CompiledCodeSignalHandler != NULL)
        p_CompiledCodeSignalHandler(sig, info, uc);
}
```

The function pointed at by this method can be used together with a clever compiler to create code in which normal flow-of-control is handled more efficiently, at the expense of making error handling much slower. For example, a SPARC implementation could ensure that a register is nonnull using the code sequence

```
cmp     <reg>, %g0
teq     16
```

This code sequence causes an interrupt if the register contains a null value. Similarly, the code for array-bounds checking could be

```
cmp     <index>, <upper-bound>
tgeu    17
```

This code sequence causes an interrupt if the register contains a value that is negative, or is greater than or equal to the upper bound. The signal handler code would be responsible for recognizing the interrupt as being an exception, and cleaning up by causing the appropriate Java `Exception` to be thrown.

5.7 (void *)p_ReadInCompiledCode(. . .), char *CompiledCodeAttribute

The variable `CompiledCodeAttribute` and the function `ReadInCompiledCode` are used by pre-compilers to read in native code from "fat" class files that contain both native code and Java byte codes.

If the value of the variable `CompiledCodeAttribute` is not `NULL`, it is considered to be a string giving the name of the attribute that specifies native code in the class file. The string should hardware-specific. The *Java Virtual Machine Specification* suggests that ompiler writers should make it vendor- or application-specific.

Normally, the code that parses a class file ignores any attributes in the class file that it does not recognize. However, if the following three conditions are met:

- The value of `CompiledCodeAttribute` is not null.
- The name of a method's attribute matches the value of the variable `CompiledCodeAttribute`, then the attribute is assumed to give the native-language implementation of the method.

---

* Currently, this method is only called in Java implementations on Solaris.

- The current class has not been loaded in through a class loader.*

The function `ReadInCompiledCode` is called to read the compiled code for that method. It is called with the following seven arguments:

- `(void *)context`: The compiler need not look at this variable. It should be passed back as the first argument to `get1byte`, `get2bytes`, `get4bytes`, and `getNbytes`.
- `int attribute_length`: The length of the `CompiledCodeAttribute` attribute.
- `struct methodblock *mb`: The method on which the compiled code attribute was found. The newly read compiled code can be attached to the `mb` argument, for example, by making it the value of the `mb->CompiledCode` field.
- `unsigned long (*get1byte)(void *context)`: The compiler should call this function with the `context` as its single argument, to read the next byte of the attribute.
- `unsigned long (*get2bytes)(void *context)`: The compiler should call this function with the `context` as its single argument, to read the next two bytes of the attribute as an unsigned value in big-endian order.
- `unsigned long (*get4bytes)(void *context)`: The compiler should call this function with the `context` as its single argument, to read the next four bytes of the attribute as an unsigned value in big-endian order.
- `void (*getNbytes)(void *, int count, char *buffer)`: The compiler should call this function with `context`, a `count`, and a buffer. The function reads the next `n` bytes of the attribute and puts it into the buffer. If the `buffer` argument is `NULL`, `n` bytes of the attribute are skipped.

The definition of `ReadInCompiledCode` is as follows:

```
void ReadInCompiledCode(void *context, struct methodblock *mb,
                        int attribute_length,
                        unsigned long (*get1byte)(),
                        unsigned long (*get2bytes)(),
                        unsigned long (*get4bytes)(),
                        void (*getNbytes)())
{
    if (p_ReadInCompiledCode != NULL) {
        p_ReadInCompiledCode(context, mb, attribute_length,
                             get1byte, get2bytes, get4bytes,
                             getNbytes);
    } else {
        getNbytes(context, attribute_length, NULL);
    }
}
```

The function pointed at by `p_ReadInCompiledCode` must read (or skip) a total of exactly `attribute_length` bytes of data; otherwise the results are unpredictable.

---

* For security reasons, the current implementation of the JDK does not accept native code from classes loaded via a class loader. This restriction is because the JDK has no way of verifying native code. In future releases, this restriction may be relaxed for signed classes.

Note that if p_ReadInCompiledCode is 0, the getNbytes function is called to skip attribute_length bytes of input.

5.8 boolean (*p_PCinCompiledCode)(unsigned char *pc, struct methodblock *mb) (unsigned char*)(*p_CompiledCodePC)(JavaFrame *frame, struct methodblock *mb) JavaFrame *(*p_CompiledFramePrev)(JavaFrame *frame, JavaFrame *buf)

When an object of type Throwable is created, one of its fields is filled with the program counters (PC's) of all the frames above it in the stack.

Three hooks are provided for dealing with stack frames, methods, and program counters

- CompiledCodePC determines the program counter of a compiled frame
- CompiledFramePrev determines the previous frame of a compiled frame.
- PCinCompiledCode determines if a previously determined program counter is within a compiled method.

The Java virtual machine determines the program counter of a specific frame as follows:

- If a specific frame indicates that it is running an interpreted (non-compiled) method, the PC is extracted directly from the frame.
- If the specific frame indicates that it is running a compiled method, the PC is extracted by calling CompiledCodePC function with the appropriate frame and method block

The definition of CompiledCodePC is as follows:

```
unsigned char *
CompiledCodePC(JavaFrame *frame, struct methodblock *mb) {
    return (p_CompiledCodePC == NULL)
                ? NULL
                : p_CompiledCodePC(frame, mb);
}
```

To walk up the stack, the Java Virtual Machine also looks at the current frame:

- If a specific frame indicates that it is running an interpreted (non-compiled) method, the previous frame is extracted directly from the frame by frame->prev.
- If the specific frame indicates that it is running a compiled method, the previous frame is extracted by calling CompiledFramePrev(frame, &temp_buffer).temp_buffer is a JavaFrame structure whose value must not be modified by the caller. Its contents must be passed unchanged to the next call to CompiledFramePrev as the code walks up the stack.

In general, code to walk up the Java stack looks like the following:

```
JavaFrame *frame = ee->current_frame;
JavaFrame frame_buf;
```

```
while (frame != NULL) {
    if (start_frame->current_method != NULL) {
        < do something >
    }
    if (frame->current_method->fb.access &
                ACC_MACHINE_COMPILED) {
        frame = CompiledFramePrev(frame, &frame_buf)
    } else {
        frame = frame->prev;
    }
}
```

The definition of `CompiledFramePrev` is as follows:

```
JavaFrame *CompiledFramePrev(JavaFrame *frame, JavaFrame *buf) {
    return (p_CompiledFramePrev == 0)
        ? frame->prev
        : p_CompiledFramePrev(frame, buf);
}
```

Note that if `p_CompiledFramePrev` is zero, the previous frame of a compiled frame is calculated in exactly the same manner as the previous frame of an interpreted frame.

The function `PCinCompiledCode` is used to determine if a specific program counter is within the bounds of a specific method. The PC argument is either a value inside the code of some interpreted method, or is a value returned by `CompiledCodePC`.

The definition of `PCinCompiled` code is as follows:

```
bool_t PCinCompiledCode(unsigned char *pc, struct methodblock *mb)
{
    return (p_PCinCompiledCode != NULL)
            && p_PCinCompiledCode(pc, mb);
}
```

Note that if `p_PCinCompiledCode` is zero, this function always returns `FALSE.`

5.9 boolean (*p_invokeCompiledMethod)(. . .)

The invoker `invokeCompiledMethod` is attached to any method that has compiled code. As a result, this function is called whenever an interpreted method invokes a method that has compiled code.

This method takes four arguments:

- `JHandle *o`: The “this” of the method call. For class (static) methods, this argument contains the handle of the class to which `mb` belongs.
- `struct methodblock *mb`: The method being invoked.
- `int args_size`: The number of arguments with which the method is being called. For instance (non-static) methods, this count includes the “this” argument.
- `ExecEnv *ee`: The current execution environment.

The currently executing frame can be found at `ee->current_frame`. The i'th argument to the current method can be found at `ee->current_frame->optop[i]`. The "this" argument is at `ee->current_frame->optop[0]`.

If the compiled code returns an integer, float, or object value, it should increment the value of `ee->current_frame->optop` by 1* and place the result at the new value of `ee->current_frame->optop[0]`.

If the compiled code returns a long or double value, it should increment the value of `ee->current_frame->optop` by 2 and place the two words of result at the new `ee->current_frame->optop[0]` and `ee->current_frame->optop[1]`. The order of the two words of the long or double is implementation dependent.†

The definition of `invokeCompiledMethod` is as follows:

```
bool_t
invokeCompiledMethod(JHandle *o, struct methodblock *mb,
                     int args_size, ExecEnv *ee)
{
    if (p_invokeCompiledMethod == NULL) {
        SignalError(ee, JAVAPKG "InternalError",
                    "Error! Compiled methods not supported");
        return FALSE;
    } else {
        return p_invokeCompiledMethod(o, mb, args_size, ee);
    }
}
```

If the value of the variable `p_invokeCompiledMethod` is not 0, it should point to a function that will cause the indicated method's compiled code to be executed with the indicated arguments. The invoker should return a non-zero value if the method executed without error. A zero value indicates that an error occurred.

If an error occurred, the fields `ee->exceptionKind` and `ee->exception` should be filled in with appropriate values giving the error.

6 Other Values in the Link Vector

In addition to the hooks in link vector given in §5, the link vector contains pointers to a wide variety of variables and functions that the compiler may want to use.

---

* That is, "increment by 1" in the C sense. The value is actually incremented by the size of one element of the stack.

† Future implementations of the Java Native Code API should address this problem. One possible solution is the creation of two new functions `returnLongToFrame` and `returnDoubleToFrame` which will increment the optop and place the two words of the long or double correctly on the stack.

On all current implementations, the long or double is placed on the stack as if by the code

```
((long *)(&ee->current_frame->optop[0])) = <long value>
((double *)(&ee->current_frame->optop[0])) = <double value>
```

However, `ee->current_frame->optop[0]` is not guaranteed to be 8-byte aligned.

6.1 Version

The link vector contains the address of the variable `JavaVersion`. Bits 24–31 of this variable contain the compiler version, bits 16–23 contain the major version of the java class file, and bits 0–15 contain the minor version of the java class file.

Currently, the compiler version is 2, the major version is 43, and the minor version is 3.

6.2 UseLosslessQuickOpcodes

The link vector contains a pointer to the variable `UseLosslessQuickOpcodes`. This variable is more fully described in §4.1

6.3 Important Class Files

The link vector contains the addresses of variables whose value is the internal class structures for the following classes and interfaces:

Link vector entry	corresponding class
`classJavaLangClass`	`class java.lang.Class`
`classJavaLangObject,`	`class java.lang.Object`
`classJavaLangString,`	`class java.lang.String`
`classJavaLangThrowable,`	`class java.lang.Throwable`
`classJavaLangException,`	`class java.lang.Exception`
`classJavaLangRuntimeException,`	`class java.lang.RuntimeException`
`interfaceJavaLangCloneable,`	`interface java.lang.Cloneable`

6.4 Generic memory allocation

The link vector contains the addresses of the functions `sysMalloc`, `sysCalloc`, `sysRealloc`, and `sysFree`. These four functions are implementations of the C library routines `malloc`, `calloc`, `realloc`, and `free`. However these implementations are guaranteed to integrated with Java's memory allocation scheme.

The compiler should use these functions for its memory allocation.

6.5 Invokers

The link vector contains the addresses of several of the invokers that can be attached to a method. Each of these methods is called with the same four arguments and stack arrangement as `invokeCompiledMethod`, which is described in §5.9.

- `invokeJavaMethod`

  This invoker is used to call an unsynchronized, interpreted Java method.

- `invokeSynchronizedJavaMethod`

  This invoker is used to call a synchronized, interpreted Java method.

- `invokeAbstractMethod`:

  This invoker is used to call an abstract or otherwise unimplemented method. It generates an error.

- `invokeLazyNativeMethod`

  This invoker is used to call a native method whose address is not yet known. The first time that this invoker is called, it calls `dynoLink` (see below) on the method block to link in the native method (if necessary) and get its address. It then changes the `mb->invoker` field of its method to be either `invokeSynchronizedNativeMethod` or `invokeNativeMethod`, depending on whether the method was originally declared synchronized or not.

- `invokeSynchronizedNativeMethod`*

  This invoker is used to call synchronized native methods.

- `invokeCompiledMethod`

  This invoker is used to call compiled methods.

The native-language implementation of native methods are not necessarily loaded at the same time that the class file is. The function `dynoLink` is called the first time a specific native method is invoked

- `bool_t dynoLink(struct methodblock *mb)`

  This function is called to dynamically link in the address of a native method. It should only be called if `mb` is a native method that has not yet been linked. If this function succeeds, it places the address of the native code into `mb->code` and returns `TRUE`. Otherwise, it returns `FALSE`.

6.6 Monitors

The Java Virtual Machine provides two ways of creating and referencing monitors, by using a "key", or by directly creating and initializing a monitor.

When creating a monitor using a key, any unsigned integer value can be used as the key. This is the sort of monitor used by the current JVM for synchornized methods. These monitors are created when needed, and then automatically discarded as necessary, when no longer used.

A monitor can also be directly created and initialized. Such monitors tend to be faster than keyed monitors, since there is no table-lookup associated with them. However these monitors are not reclaimed and exist for the life of the program.

The link vector contains the addresses of six functions useful for manipulating monitors:

- `void monitorEnter(unsigned int key)`

* For unkown reasons, the address of `invokeNativeMethod`, which is used to invoke non-synchronized native methods is not included in the link vector. No one seems to have noticed.

Creates a monitor associated with the specified key, if such a monitor doesn't already exist. The function then tries to enter the monitor using `sysMonitorEnter`.

The current JVM, when entering a synchronized instance (non-static) method, attempts to lock "this" by calling `monitorEnter((unsigned int) this)`. When entering a synchrnized class (static) method, it locks the class by calling `monitorEnter((unsigned int)<`*`handle to class Object`*`>)`. The corresponding `monitorExit` function is called when leaving the synchronized method.

- `void monitorExit(unsigned int key)`

  Finds the monitor associated with the specified key (it must already exist) and then exits that monitor using `sysMonitorExit`.

- `void monitorRegister(sys_mon_t *mid, char *name)`

  Creates and registers a new monitor. The `mid` argument must point to a buffer whose size is at least as big as that returned by `sysMonitorSizeof()`. The new monitor is given the specified name. This name is used by various debugging routines (c.f. `DumpThreads()` in §6.10) which print out the state of the machine.

- `void sysMonitorEnter(sys_mon_t *mid)`

  If the monitor indicated by the `mid` argument is unowned or already owned by the current thread, then execution proceeds normally. Otherwise, the current thread waits for the monitor to become free. Note that a count is kept of how many times the current thread has "entered" the monitor.

  The monitor argument `mid` must either be a monitor created automatically by `monitorEnter`, or it must be registered through a call to `monitorRegister`. Failing to register a monitor before using it may cause unpredictable results.

- `void sysMonitorExit(sys_mon_t *mid)`

  The monitor indicated by the `mid` argument must be owned by the current thread. The count indicating how many times the current thread has "entered" the monitor is decremented. If the count goes to zero, the current thread gives up ownership of the monitor.

  The monitor argument `mid` must either be a monitor created automatically by `monitorEnter`, or it must be registered through a call to `monitorRegister`. Failing to register a monitor before using it may cause unpredictable results.

  An error is signalled if `sysMonitorExit` is called by a thread that does not currently own the monitor.

- `int sysMonitorSizeof()`

  Returns the size of the `sys_mon_t` structure.

6.7 Class access

The link vector contains the addresses of two variables and two functions for finding all classes that have already been loaded directly:

```
int nbinclasses;
ClassClass** binclasses;
void *lockClasses();
void *unlockClasses()
```

The variable binclasses points to an array of all the classes that have been loaded into the system. The number of elements currently in this array is given by nbinclasses.

No code should not look at or modify binclasses and nbinclasses without first calling the function lock_classes(). Calling this function acquires a monitor which is set aside for these variables. When the code is finished modifying these variables, it should free the monitor by calling unlock_classes().

The link vector also contains the addresses of two functions by which the compiler can find a class by name. If the specified class has already been loaded, it is returned. If the class has not yet been loaded, it is loaded.

```
ClassClass
  *FindClass(struct execenv *ee, char *name, bool_t resolve);
ClassClass ClassClass
  *FindClassFromClass(struct execenv *ee, char *name,
                      bool_t resolve, ClassClass *from);
```

The function FindClass is a convenience function. It looks at the current execution environment to determine the class of the currently running method. It then calls FindClassFromClass with the identical arguments that it was passed, but adds the current class as the from argument.

The function FindClassFromClass is the general function for loading and finding classes.

- If the from class does not have a class loader, then binclasses is searched to see if a class with the specified name has already been loaded. If not, the class is loaded.
- If the from class does have a class loader, the class loader is called to determine if it has already loaded in the class, or if a new class needs to be loaded.

6.8 Object Allocation

The link vector contains pointers to the following functions which are used to allocate instances and arrays.

- HObject *ObjAlloc(ClassClass *cb, long n):

  The first argument must point to the class block of a non-array class. The second argument must currently always be 0.

  A space is allocatyped for the object, but the space is not initialized in any way. The contents of the space is unpredictable.

- HObject *newobject(ClassClass *cb,
                      unsigned char *pc, struct execenv *ee)

  This method allocates space for the specified object by calling ObjAlloc, and then zeroes the space. If any error occurs, SignalError is called with appropriate arguments.

- HObject *ArrayAlloc(int type, int length)

Space is allocated for an array of the specified type and length. The possible values for type are shown in the table below:

Array Type	*atype*
`T_CLASS`	2
`T_BOOLEAN`	4
`T_CHAR`	5
`T_FLOAT`	6
`T_DOUBLE`	7
`T_BYTE`	8
`T_SHORT`	9
`T_INT`	10
`T_LONG`	11

The arrays are not initialized, except that for arrays of type T_CLASS, the slot that contains the class type is initialized to 0.

- `sizearray(int type, int length).`

  The method returns the size of an array of the specified type and length. The type must be one of the arguments given in the table above for `ArrayAlloc`. Note that when `type` is `T_CLASS`, the value returned by `sizearray` does *not* include the extra word at the end indicating to the type.*

- `HObject *MultiArrayAlloc(int dimensions, ClassClass *array_cb, stack_item *sizes)`

  This function is used to allocate several dimensions of a multi-dimensional array simultaneously. The dimensions given by the dimensions `argument` must be less than or equal to the dimensions of the array class block specified by the `array_cb` argument.

  For example, to create `new int[5][4][]`, `MultiArrayAlloc` would be called with

```
dimensions = 2
array_cb = FindClass("[[I", TRUE)
sizes[0] = 5;
sizes[1] = 4;
```

6.9 Constant Pool Resolution

The link vector contains the addresses of several functions that are used to resolve entries in a class's constant pool.

- ```
  bool_t
  ResolveClassConstantFromClass(ClassClass *class,
                                unsigned index, struct execenv *ee,
                               unsigned mask)
  ```

* In retrospect, this is probably a mistake.
  ```

The `index`'th entry in `class`'s constant pool is resolved if it has not already been resolved. If the entry has not yet been resolved, its type must match one of the types specified by the bitmask given in `mask`. The possible types are given in the following table:

Constant Type	**Value**
`CONSTANT_Class`	7
`CONSTANT_Fieldref`	9
`CONSTANT_Methodref`	10
`CONSTANT_InterfaceMethodref`	11
`CONSTANT_String`	8
`CONSTANT_Integer`	3
`CONSTANT_Float`	4
`CONSTANT_Long`	5
`CONSTANT_Double`	6
`CONSTANT_NameAndType`	12
`CONSTANT_Utf8`	1

If the `index`'th entry is of type `x`, then the value of

```
mask & (1 << x)
```

must not be zero. Specifying a mask of `((unsigned)-1)` ensures that any constant pool type can be resolved. More information on the constant pool and its types can be found in *The Java Virtual Machine*.

All entries in the constant pool are resolved with respect to the `class` argument. Any references to other classes are resolved by calling `FindClassFromClass` using the class as the `from` argument.

This function returns `TRUE` if the constant pool entry was successfully resolved; `FALSE` otherwise.

- ```
  bool_t
  ResolveClassConstant(cp_item_type *constant_pool,
                        unsigned index, struct execenv *ee,
                       unsigned mask)
  ```

  This function is similar to `ResolveClassConstantFromClass`. However the constant pool is given directly, rather than being inferred from the `class` argument.

  If the current execution environment's current frame has a current method, then that method's class is used for resolving any other class references in the constant pool. If the current frame has no current method, then other class reference are presumed to reside locally.

  This function returns `TRUE` if the constant pool entry was successfully resolved; `FALSE` otherwise.

- ```
  GetClassConstantClassName(cp_item_type *constant_pool,
                            unsigned index)
  ```

  This function can be used to get the name of the class or interface at the `index`'th item in the specified constant pool. This function does not resolve the entry at the constant pool.

- ```
  bool_t
  VerifyClassAccess(ClassClass *current_class,
                     ClassClass *new_class,
                    bool_t classloaderOnly)
  ```

  This function determines if the specified `current_class` is allowed to access the specified `new_class`. If `current_class` is `NULL`, this function always returns `TRUE`.

  If the `current_class` has no class loader, and the `classloaderOnly` flag is `TRUE`, this method also always returns `TRUE`.*

  Otherwise, a class can only access another class if at least one of the following conditions is true:

  - They are the same class.
  - The second class is public.
  - The two classes are in the same package.

6.10 Miscellaneous Runtime Functions and Variables

The following are a set of miscellanous funcions and variables whose addresses are in the link vector.

- `struct execenv *EE()`

  This function returns the execution environment for the current thread.

- ```
  void SignalError(struct execenv *ee, char *ename,
                   char *DetailMessage)
  ```

  This function modifies the execution environment indicated by the `ee` argument to indicate than an error has occurred. The `ename` argument must be the name of a subclass of `Throwable`.

  If the `ee` argument is `NULL`, the value of `EE()` is used instead.

- `exception_t exceptionInternalObject(int i)`

  Calling `exceptionInternalObject(1)` returns a pre-allocated object of type `NoClassDefFoundError` object.

  Calling `exceptionInternalObject(2)` returns a pre-allocated object of type `OutOfMemoryError` object

- ```
  bool_t is_subclass_of(ClassClass *cb, ClassClass *dcb,
                        struct execenv *ee)
  ```

  The argument `cb` must point to class structure. The argument dcb must point to either a class or to an interface structure. This function returns `TRUE` if the class `cb` is a subclass of or implements the interface `dcb`.

- ```
  bool_t is_instance_of(JHandle *o, ClassClass *dcb,
                        struct execenv *ee)
  ```

  This function returns `TRUE` if the instance `o` is a subclass of or implements the interface `dcb`. This method always returns `TRUE` if `o` is the `null` object.

---

* This misfeature is to work around a bug in the current Java compiler. Code compiled “-O” can inline calls to methods to which the class shouldn’t normally have access. This will be fixed in a future release.

- `classname2string`

  Not yet written.

- `void DumpThreads()`

  This function is useful for debugging. It prints out a full stack trace of every running thread in the Java Virtual machine.

- `execute_java_static_method`

  Not yet written.

- `long now()`

  This function returns the current time in milliseconds.

- `bool_t java_monitor`

  If this variable is non-zero, it indicates that we are currently monitoring and profiling method execution.

- ```
  void java_mon(struct methodblock *caller,
                 struct methodblock *callee,
                int time)
  ```

  If the variable `java_monitor` is non-zero, this function should be called at the end of every method execution. The three arguments are the calling method, the called method, and the total amount of time spent in the called method.

- `int jio_snprintf(char *buf, int len, char *fmt, ...)`

  This function is identical to the C library routine `sprintf`, except that it takes an additional `len` argument giving the length of the buffer. The formatted arguments are written to the buffer, but at most len bytes are written. The value is the buffer is always `null` terminated.

  The value returned is the total number of bytes that would be required to write out the specified arguments in the specified format, not including the null terminating byte.

- ```
  char *javaString2CString(Hjava_lang_String *s, char *buf,
                           int buflen)
  ```

  The specified Java string is converted to a C string, and the results are put into the specified buffer. At most `buflen` - 1 bytes are written to the buffer, followed by a `null` character.

  The `buf` argument is returned as the value

- `ID2Str`

  Not yet written.

- `interfaceHash`

  Not yet written.

6.11 Functions you don't want to touch

- `do_execute_java_method_vararg`

  Not yet written

- ```
  JavaStack CreateNewJavaStack(ExecEnv *ee,
                               JavaStack *previous_stack)
  ```
  ```

This method creates a new Java stack chunk and appends it to the previous stack chunk. The previous stack chunk must be the last chunk in its chain. The new stack chunk is created, appended to the chain, and returned.

- `bool_t ExecuteJava`(unsigned char *pc, ExecEnv *ee)

  Begin executing the Java byte codes beginning at address `pc`. The ee->frame

- `JavaStackSize`

  The value of this variable is the maximum stack size, in bytes, that a thread can use for its Java stack.

7 Additional Information