Hardware Objects for Java

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Christian Thalinger
Institute of Computer Languages
Vienna University of Technology, Austria
twisti@complang.tuwien.ac.at

Stephan Korsholm
Department of Computer Science
Aalborg University DK-9220 Aalborg
stk@cs.aau.dk

Anders P. Ravn
Department of Computer Science
Aalborg University DK-9220 Aalborg
apr@cs.aau.dk

Abstract

Java, as a safe and platform independent language, avoids access to low-level I/O devices or direct memory access. In standard Java, low-level I/O it not a concern; it is handled by the operating system.

However, in the embedded domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture. When running the JVM on bare metal*, we need access to I/O devices from Java; therefore we investigate a safe and efficient mechanism to represent I/O devices as first class Java objects, where device registers are represented by object fields. Access to those registers is safe as Java's type system regulates it. The access is also fast as it is directly performed by the bytecodes getfield and putfield.*

Hardware objects thus provide an object-oriented abstraction of low-level hardware devices. As a proof of concept, we have implemented hardware objects in three quite different JVMs: in the Java processor JOP, the JIT compiler CACAO, and in the interpreting embedded JVM SimpleRTJ.

1 Introduction

In embedded systems Java is now considered an alternative to C/C++. Java improves the safety of programs due to compile time type checking, additional runtime checks, and reference integrity. Those properties result in an increase of programmer productivity. Furthermore, Java is much more portable and thus facilitates reuse.

However, portability and safety comes at a cost: access to low-level devices (common in embedded systems) is not possible from pure Java. One has to use native functions that are implemented in C/C++. Invocation of those native functions incurs runtime overheads. Often they are developed in ad-hoc fashion, thus making them error prone as well; for instance if they interfere with the Java VM or garbage collection when addressing Java objects. Before we present our proposed solution, *Hardware Objects*, we describe the problem as seen from a Java virtual machine (JVM).

1.1 Embedded JVMs

The architecture of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 1 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach of a JVM running on top of an operating system (OS) is shown in sub-figure (a). A network connection bypasses the JVM via native functions and uses the TCP/IP stack and device drivers of the OS.

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides thread scheduling and low-level access to the hardware. Thus the stack can be written entirely in Java. An example of the approach is JNode[1] which implements the OS fully in Java.

Sub-figure (c) shows a solution where the JVM is a Java processor. With this solution the native layer can be completely avoided and all code is Java.

However, for both (b) and (c) we need access to device registers and in some applications also interrupts. Here we focus on an object-oriented approach to access device registers which is compatible with Java. The issue of interrupts is treated in a companion paper [7], because it is more related to synchronization and thread scheduling.

1.2 Related Work

An excellent overview of historical solutions to access hardware devices from high-level languages, including C, is presented in Chapter 15.2 of [2]. The solution in Modula-1 (Ch. 15.3.1) is very much like C; however the constructs are safer, because they are encapsulated in modules. In Ada (Ch 15.4.1) the representation of individual fields in registers can be described precisely using *representation* classes, while the

[1] `http://www.jnode.org/`

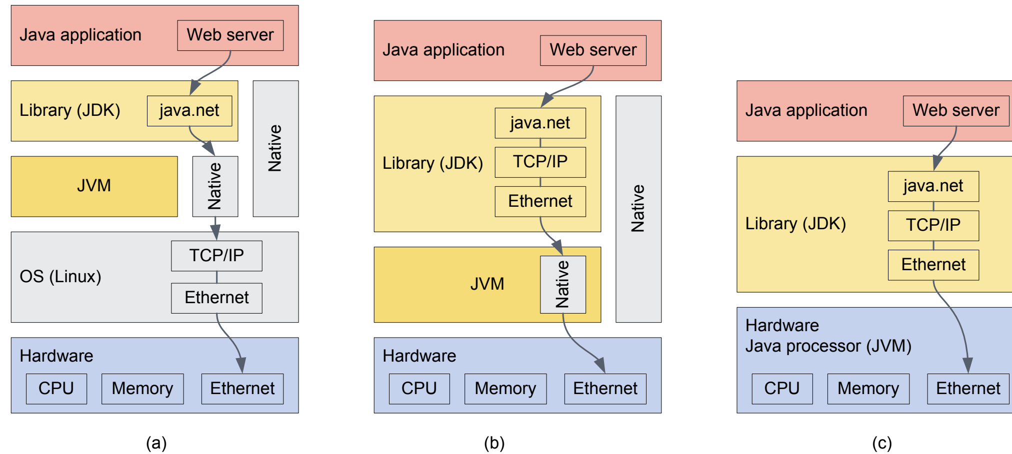

Figure 1. (a) Standard layers for embedded Java with an operating system, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

corresponding structure is bound to a location using the Address attribute.

More recently, the RTSJ [1] does not give much support. Essentially, one has to use RawMemoryAccess at the level of primitive data types. A similar approach is used in the Ravenscar Java profile [8]. Although the solution is efficient, this representation of physical memory is not object oriented and there are some safety issues: When one raw memory area represents an address range where several devices are mapped to there is no protection between them.

The aJile processor [6] uses native functions to access devices through IO pins. The Squawk VM [15], which is a JVM mostly written Java that runs without an operating system, uses device drivers written in Java. These device drivers use a form of peek and poke interface to access the device's memory. The JX Operating System [3] uses *memory objects* to provide read-only memory and device access, which are both required by an OS. Memory objects represent a region of the main address space and accesses to the regions are handled via normal method invocations on the memory objects representing the different regions.

The distinctive feature of our proposal is that it maps a hardware object onto the OO address space and provide, if desired, access methods for individual fields, such that it lifts the facilities of Ada into the object oriented world of Java.

The remainder of the paper is structured as follows: in Section 2 we motivate hardware objects and present the idea. Section 3 provides details on the integration of hardware objects into three different JVMs: a Java processor, a Just-in-time (JIT) compiling JVM, and an interpreting JVM. We conclude and evaluate the proposal in Section 4.

2 Hardware Objects

Let us consider a simple parallel input/output (PIO) device. The PIO provides an interface between I/O registers

```
typedef struct {
    int data;
    int control;
} parallel_port;
#define PORT_ADDRESS 0x10000;

int inval, outval;
parallel_port *mypp;
mypp = (parallel_port *) PORT_ADDRESS;
...
inval = mypp->data;
mypp->data = outval;
```

Figure 2. Definition and usage of a parallel port in C

and I/O pins. The example PIO contains two registers: the *data register* and the *control register*. Writing to the data register stores the value into a register that drives the output pins. Reading from the data register returns the value that is present at the input pins.

The control register configures the direction for each PIO pin. When bit *n* in the control register is set to 1, pin *n* drives out the value of bit *n* of the data register. A 0 at bit *n* in the control register configures pin *n* as input pin. At reset the port is usually configured as input port[2] – a safe default configuration.

When the I/O address space is memory mapped, such a parallel port is represented in C as a structure and a constant for the address. This definition is part of the board level configuration. Figure 2 shows the parallel port example. The parallel port is directly accessed via a pointer in C. For a system with a distinct I/O address space access to the device registers is performed via distinct machine instructions. Those instructions are represented by C functions that take the address as argument, which is not a type-safe solution.

[2]Output can result in a short circuit between the I/O pin and the external device when the logic levels are different.

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMagic.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Figure 3. The parallel port device as a simple Java class

```
package com.board-vendor.io;

public class IOSystem {

    // do some JVM magic to create the PP object
    private static ParallelPort pp = JVMPPMagic();
    private static SerialPort sp = JVMSPMagic();

    public static ParallelPort getParallelPort() {
        return pp;
    }
    public static SerialPort getSerialPort() {..}
}
```

Figure 4. A Factory with static methods for Singleton hardware objects

This simple representation of memory mapped I/O devices in C is efficient but unsafe. On a standard JVM, native functions, written in C or C++, allow low-level access to devices from Java. This approach is neither safe nor object-oriented (OO) and incurs a lot of overheads; parameters and return values have to be converted between Java and C.

In an OO language the most natural way to represent an I/O device is as an object. Figure 3 shows a class definition and object instantiation for our simple parallel port. The class ParallelPort is equivalent to the structure definition for C in Figure 2. Reference myport points to the hardware object. The device register access is similar to the C version.

The main difference to the C structure is that the access requires no pointers. To provide this convenient representation of I/O devices as objects we just need some *magic* in the JVM and a mechanism to *create* the device object and receive a reference to it. Representing I/O devices as first class objects has following benefits:

Safe: The safety of Java is not compromised. We can access only those device registers that are represented by the class definition.

Efficient: For the most common case of memory mapped I/O device access is through the bytecodes getfield and putfield; for a separate I/O address space the IO-instructions can be included in the JVM as variants of these bytecodes for hardware objects. Both solutions avoid expensive native calls.

```
public class IOFactory {

    private final static int SYS_ADDRESS = ...;
    private final static int SERIAL_ADDRESS = ...;
    private SysDevice sys;
    private SerialPort sp;
    IOFactory() {
        sys = (SysDevice) JVMIOMagic(SYS_ADDRESS);
        sp = (SerialPort) JVMIOMagic(SERIAL_ADDRESS);
    };
    private static IOFactory single = new IOFactory();
    public static IOFactory getFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return sp; }
    public SysDevice getSysDevice() { return sys; }
    // here comes the magic!
    Object JVMIOMagic(int address) {...}
}

public class DspioFactory extends IOFactory {

    private final static int USB_ADDRESS = ...;
    private SerialPort usb;
    DspioFactory() {
        usb = (SerialPort) JVMIOMagic(USB_ADDRESS);
    };
    static DspioFactory single = new DspioFactory();
    public static DspioFactory getDspioFactory() {
        return single;
    }
    public SerialPort getUsbPort() { return usb; }
}
```

Figure 5. A base class of a hardware object Factory and a Factory subclass

2.1 Hardware Object Creation

Representing the registers of each I/O device by an object or an array is clearly a good idea; but how are those objects created? An object that represents an I/O device is a typical Singleton [4]. Only one object should map to a single device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM magic to the device registers; (2) each device is represented by a single object.

One may assume that the board manufacturer provides the classes for the hardware objects and the configuration class for the board. This configuration class provides the Factory [4] methods (a common design pattern to create Singletons) to instantiate hardware objects.

Each I/O device object is created by its own Factory method. The collection of those methods is the board configuration which itself is also a Singleton (we have only one board). The configuration Singleton property is enforced by a class that contains only static methods. Figure 4 shows an example for such a class. The class IOSystem represents a minimal system with two devices: a parallel port as discussed before to interact with the environment and a serial port for program download and debugging.

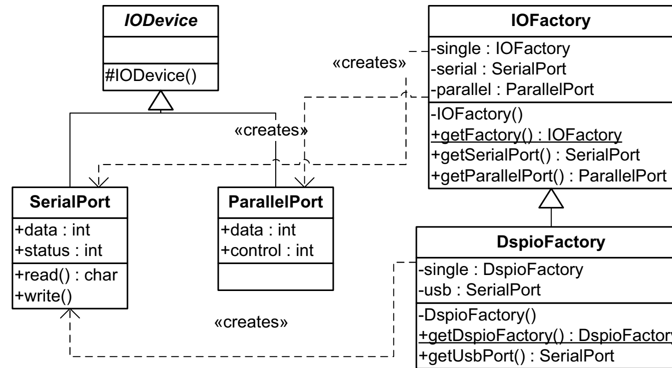

Figure 6. Hardware object classes and board Factory classes

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the approach described above each board requires a different IOSystem class that lists all devices.

2.2 Board Configurations

We can avoid the duplication of code by replacing the static Factory methods by instance methods and use inheritance for different configurations. With a Factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the Factory object itself shall still be a Singleton. Therefore the board specific Factory object is created at class initialization and can be retrieved by a static method. Figure 5 shows an example of a base Factory and a derived Factory. Note how getFactory() is used to get a single instance of the hardware object Factory. We have applied the idea of a Factory two times: the first Factory generates an object that represents the board configuration. That object is itself a Factory that generates the objects that represent the actual devices – the hardware objects.

The shown example Factory is a simplified version of the minimum configuration of the JOP [11] FPGA module *Cycore* and an extension with an I/O board that contains an USB interface.

Furthermore, we show in Figure 5 a different way to incorporate the JVM *magic* into the Factory: we define well known constants (the memory addresses of the devices) in the Factory and let the native function JVMIOMagic() return the correct I/O device type.

Figure 6 gives a summary example (a slight variation of the former example) of hardware object classes and the corresponding Factory classes as an UML class diagram. The serial port hardware object contains additional access methods to the device register fields. The figure shows that all I/O classes subclass the abstract class IODevice, a detail we have omitted in our discussion so far.

```
public class Example {

    public static void main(String[] args) {

        IOFactory fact = IOFactory.getFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit data register empty
            while ((sp.status & SerialPort.MASK_TDRE)==0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Figure 7. Hello World using hardware objects

2.3 Using Hardware Objects

Creation of hardware objects is a bit complex, but usage is very simple. After obtaining a reference to the object all what has to be done (or can be done) is to read from and write to the object fields. Figure 7 shows an example of the client code. The example is the *Hello World* program using low-level access to the terminal via a hardware object.

3 Implementations

In order to show that our proposed approach is workable we have chosen three completely different JVMs for the evaluation: a Java processor (JOP [11, 13]), a JIT JVM (CACAO [5]) and a small interpreting JVM (the SimpleRTJ VM [10]). All three projects are open-source and make it possible for us to show that hardware objects can be implemented in very different JVMs.

We provide implementation details to help other JVM developers to add hardware objects to their JVM. The techniques used for JOP, CACAO, or SimpleRTJ cannot be used one-to-one. However, the solutions (or sometimes a workaround) presented here should guide other JVM developers.

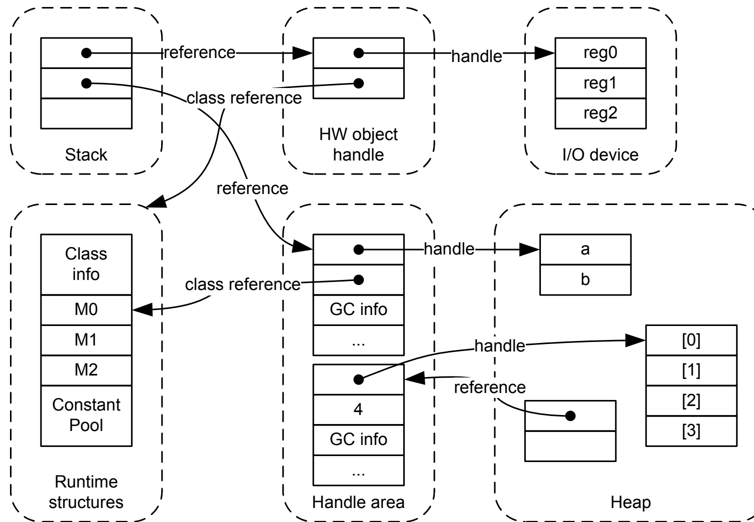

Figure 8. Memory layout of the JOP JVM

3.1 HW Objects on JOP

We have implemented the proposed hardware objects in the JVM for the Java processor JOP [11, 13]. No changes inside the JVM (the microcode in JOP) were necessary. The tricky part is the creation of hardware objects (the Factory classes).

3.1.1 Object Layout

In JOP objects and arrays are referenced through an indirection, called the *handle*. This indirection is a lightweight read barrier for the compacting real-time garbage collector (GC) [12, 14]. All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 8 shows an example with a small object that contains two fields and an integer array of length 4. We can see that the object and the array on the heap just contain the data and no additional hidden fields. This object layout greatly simplifies our object to I/O device mapping. We just need a handle where the indirection points to the memory mapped device registers. This configuration is shown in the upper part of Figure 8. Note that we do not need the GC information for the HW object handles.

3.1.2 The Hardware Object Factory

As described in Section 2.1 we do not allow applications to create hardware objects; the constructor is private. Two static fields are used to store the handle to the hardware object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information. The address of the first static field is returned as the reference to the serial port object. We have to solve two issues: (1) obtain the class reference for the HW object; (2) return the address of a static field as a reference to the hardware object.

We have two options to get a pointer to the class information of a hardware object, such as SerialPort, in a method of IOFactory:

1. Create a normal instance of SerialPort with new on the heap and copy the pointer to the class information.

2. Invoke a static method of SerialPort. The method executes in the context of the class SerialPort and has access to the constant pool of that class and the rest of the class information.

Option 1 is simple and results in following code for the object factory:

```
SerialPort s = new SerialPort();
int ref = Native.toInt(s);
SP_MTAB = Native.rdMem(ref+1);
```

All methods in class Native, a JOP system class, are *native*[3] methods for low-level functions – the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method rdMem(int addr) performs a memory read. In our case the second word from the handle, the pointer to the class information. The main drawback of option 1 is the creation of normal instances of the hardware class. With option 1 the visibility of the constructor has to be relaxed to package.

For option 2 we have to extend each hardware object by a class method to retrieve the address of the class information. Figure 9 shows the version of SerialPort with this method. We use again native functions to access JVM internal information. In this case rdIntMem(1) loads one word from the

[3]There are no native functions in JOP – bytecode is the native instruction set. The very few native functions in class Native are replaced by a special, unused bytecode during class linking.

```
public final class SerialPort {

    public volatile int status;
    public volatile int data;

    static int getClassRef() {
        // we can access the constant pool pointer
        // and therefore get the class reference
        int cp = Native.rdIntMem(1);
        ...
        return ref;
    }
}
```

Figure 9. A static method to retrieve the address of the class information

on-chip memory onto the top-of-stack. The on-chip memory contains the stack cache and some JVM internal registers. At address 1 the pointer to the constant pool of the actual class is located. From that address we can calculate the address of the class information. The main drawback of option 2 is the repetitive copy of getClassRef() in each hardware class. As this method has to be static (we need it before we have an actual instance of the class) we cannot move it to a common superclass.

We decided to use option 1 to avoid the code duplication. The resulting package visibility of the hardware object constructor is a minor issue.

All I/O device classes and the Factory classes are grouped into a single package, in our case in com.jopdesign.io. To avoid exposing the native functions (class Native) that reside in a system package we use delegation. The Factory constructor delegates all low-level work to a helper method from the system package.

3.2 HW Objects in CACAO

As a second experiment we have implemented the hardware objects in the CACAO VM [5]. The CACAO VM is a research JVM developed at the Vienna University of Technology and has a Just-In-Time (JIT) compiler for various architectures.

3.2.1 Object layout

As most other JVMs, CACAO's Java object layout includes an object header which is part of the object itself and resides on the garbage collected heap (GC heap). This fact makes the idea of having a *real* hardware-object impossible without changing the CACAO VM radically. Thus we have to use an indirection for accessing hardware-fields and hardware-arrays. Having an indirection adds obviously an overhead for accesses to hardware-fields or hardware-arrays. On the other hand, CACAO does widening of primitive fields of the type boolean, byte, char, and short to int which would make it impossible to access hardware-fields smaller than int directly in a Java object. With indirection we can solve this issue. We store the address of the hardware-field in a Java object field and access the correct data size in JIT code.

When it comes to storing the hardware address in a Java object field, we hit another problem. CACAO supports 32 and 64-bit architectures and obviously a hardware address of a byte-field on a 64-bit architecture won't fit into a widened 32-bit object field. To get around this problem we widen all object fields of sub-classes of org.cacaovm.io.IODevice to the pointer size on 64-bit machines. To be able to widen these fields and to generate the correct code later on in the JIT compiler, we add a VM internal flag ACC_CLASS_HARDWARE_FIELDS and set it for the class org.cacaovm.io.IODevice and all its subclasses, so the JIT compiler can generate the correct code without the need to do super-class tests during the JIT compiler run. For hardware-arrays we have to implement a similar approach. The object layout of an array in CACAO looks like this:

```
typedef struct java_array_t {
    java_object_t objheader;
    int32_t       size;
} java_array_t;

typedef struct java_intarray_t {
    java_array_t header;
    int32_t      data[1];
} java_intarray_t;
```

The data field of the array structure is expanded to the actual size when the array object is allocated on the Java heap. This is a common practice in C.

When we want to access a hardware array we have the same problem as for fields – the array header. We cannot put the array directly on the hardware addresses. Therefore we add a union to the java_xxxarray_t-structures:

```
typedef struct java_intarray_t {
    java_array_t header;
    union {
        int32_t  array[1];
        intptr_t address;
    } data;
} java_intarray_t;
```

Now we can allocate the required memory for Java arrays or store the hardware address for hardware arrays into the array object.

3.2.2 Implementation

CACAO's JIT compiler generates widened loads and stores for getfield and putfield instructions. But when we want to load byte or short fields from a hardware object we need to generate 8-bit or 16-bit loads and stores, respectively. To get these instructions generated we implement additional cases in the JIT compiler for the various primitive types.

Whether the JIT compiler needs to generate 8-bit or 16-bit loads and stores for boolean, byte, char, or short fields is decided on the flags of the declared class.

Contrary to hardware fields, when accessing hardware arrays we have to generate some dedicated code for array accesses to distinguish between Java arrays and hardware arrays at runtime and generate two different code paths, one to access Java arrays and the other to access hardware arrays.

3.3 HW Objects in SimpleRTJ

In a third experiment we have implemented hardware objects for the SimpleRTJ interpreter [10]. The SimpleRTJ VM is described in more detail in [9]. To support the direct reading and writing from/to raw memory we introduced an additional version of the put/get-field bytecodes. We changed the VM locally to use these versions at bytecode addresses where access to hardware objects is performed. The original versions of put/get-field are not changed and are still used to access normal Java object fields.

The new versions of put/get-field to handle hardware objects are different. An object is identified as a hardware object if it inherits from the base class IODevice. This base class defines one 32 bit integer field called address. During initialization of the hardware object the address field variable is set to the absolute address of the device register range that this hardware object accesses.

The hardware object specific versions of put/get-field calculates the offset of the field being accessed as a sum of the width of all fields preceding it. In the following example control has an offset of 0, data an offset of 1, status an offset of 3 and finally reset an offset of 7.

```
class DummyDevice extends IODevice {
   public byte control;
   public short data;
   public int status;
   public int reset;
}
```

The field offset is added to the base address as stored in the super class instance variable address to get the absolute address of the device register or raw memory to access. The width (or number of bytes) of the data to access is derived from the type of the field.

To ensure that the speed by which normal objects are accessed do not suffer from the presence of hardware objects we use the following strategy: The first time a put/get-field bytecode is executed a check is made if the objects accessed is a hardware object. If so, the bytecode is substituted with the hardware object specific versions of put/get-field. If not the bytecode is substituted with the normal versions of put/get-field.

For this to be sound, a specific put/get-field instruction is never allowed to access both normal and hardware objects. In a polymorphic language like Java this is in general not a sound assumption. However, with the inheritance hierarchy of hardware object types this is a safe assumption.

3.4 Summary

We have described the implementation of hardware objects on JOP in great detail and outlined the implementation in CACAO and in SimpleRTJ. Other JVMs use different structures for their class and object representations and the presented solutions cannot be applied directly. However, the provided details give guidelines for changing other JVMs to implement hardware objects.

On JOP all the code could be written in Java,[4] it was not necessary to change the microcode (the low-level implementation of the JVM bytecodes in JOP). Only a single change in the runtime representation of classes proved necessary. The implementation in CACAO was straightforward. Adding a new internal flag to flag classes which contain hardware-fields and generating slightly more code for array accesses, was enough to get hardware objects working in CACAO.

4 Conclusion

We have introduced the notation of hardware objects. They provide an object-oriented abstraction of low-level devices. They are first class objects providing safe and efficient access to device registers from Java.

To show that the concept is practical we have implemented it in three different JVMs: in the Java processor JOP, in the research VM CACAO, and in the embedded JVM SimpleRTJ. The implementation on JOP was surprisingly simple – the coding took about a single day. The changes in the JIT JVM and in the interpreter JVM have been slightly more complex.

The proposed hardware objects are an important step for embedded Java systems without a middleware layer. Device drivers can be efficiently programmed in Java and benefit from the same safety aspects as Java application code.

4.1 Performance

Our main objective for hardware objects is a clean OO interface to I/O devices. Performance of the access of device registers is an important secondary goal, because short access time is important on relatively slow embedded processors while it matters less on general purpose processors, where the slow I/O bus essentially limits the access time. In Table 1 we compare the access time to a device register with native functions to the access via hardware objects.

On JOP the native access is faster than using hardware objects. This is due to the fact that a native access is a special bytecode and not a function call. The special bytecode accesses memory directly, where the bytecodes putfield and getfield perform a null pointer check and indirection through the handle.

The performance evaluation with the CACAO JVM has been performed on a 2 GHz x86_64 machine under Linux

[4] except the already available primitive native functions

	JOP		CACAO		SimpleRTJ	
	read	write	read	write	read	write
native	8	9	24004	23683	2588	1123
HWO	21	24	22630	21668	3956	3418

Table 1. Access time to a device register in clock cycles

with reads and writes to the serial port. The access via hardware objects is slightly faster (6% for read and 9% for write, respectively). The kernel trap and the access time on the I/O bus dominate the cost of the access in both versions. On an experiment with shared memory instead of a real I/O device the cost of the native function call was considerable.

On the SimpleRTJ VM the native access is slightly faster than access to hardware objects. The reason is that the SimpleRTJ VM does not implement JNI, but has it's own proprietary, more efficient, way to invoke native methods. It does this very efficiently using a pre-linking phase where the invokestatic bytecode is instrumented with information to allow an immediate invocation of the target native function. On the other hand, hardware object field access needs a field lookup that is more time consuming than invoking a static method.

4.2 Safety and Portability Aspects

Hardware objects map object fields to the device registers. When the class that represents an I/O device is correct, access to the low-level device is safe – it is not possible to read from or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

It is obvious that hardware objects are platform dependent, after all the idea is to have an interface to the bare metal. Nevertheless, hardware objects give device manufacturers an opportunity to supply supporting software that fits into Java's object-oriented framework and thus cater for developers of embedded software.

4.3 Interrupts

Hardware objects are a vehicle to write device drivers in Java and benefit from the safe language. However, most device drivers also need to handle interrupts. We have not covered the topic of writing interrupt handlers in Java. This topic is covered by a companion paper [7], where we discuss interrupt handlers implemented in Java. Jointly Java hardware objects and interrupt handlers makes it attractive to develop platform dependent middleware fully within an object-oriented framework with excellent structuring facilities and fine grained control over the unavoidable unsafe facilities.

Acknowledgement

We thank the anonymous reviewers for their detailed and insightfully comments that helped to improve the paper.

References

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[3] M. Felser, M. Golm, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, 2002.

[4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[5] R. Grafl. CACAO: A 64-Bit JavaVM Just-in-Time Compiler. Master's thesis, Vienna University of Technology, 1997.

[6] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[7] S. Korsholm, M. Schoeberl, and A. P. Ravn. Java interrupt handling. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[8] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[9] E. Potratz. A practical comparison between Java and Ada in implementing a real-time embedded system. In *SigAda '03: Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 71–83. ACM Press, 2003.

[10] RTJComputing. http://www.rtjcom.com. Visited June 2007.

[11] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[12] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[13] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Article in press and online: Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[14] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[15] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM Press, 2006.