A Hardware Abstraction Layer in Java

MARTIN SCHOEBERL
Vienna University of Technology, Austria
STEPHAN KORSHOLM
Aalborg University, Denmark
TOMAS KALIBERA
Purdue University, USA
and
ANDERS P. RAVN
Aalborg University, Denmark

Embedded systems use specialized I/O devices to interact with their environment, and since they have to be dependable, it is attractive to use a modern, type-safe programming language like Java to develop programs for them. However, standard Java, as a platform independent language, delegates access to I/O devices, direct memory access and interrupt handling to some underlying operating system or kernel, but in the embedded systems domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture. The contribution of this paper is a proposal for Java packages with hardware objects and interrupt handlers that interfaces to such a JVM. We provide implementations of the proposal directly in hardware, directly as extensions of standard interpreters, and finally with an operating system middleware. The latter solution is mainly seen as a migration path allowing Java programs to coexist with legacy system components. An important aspect of the proposal is that it is compatible with the Real-Time Specification for Java (RTSJ).

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.3 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Input/output*

General Terms: Languages, Design, Implementation

Additional Key Words and Phrases: Device driver, embedded system, Java, Java virtual machine

1. INTRODUCTION

When developing software for an embedded system, for instance an instrument, it is necessary to control specialized hardware, for instance a heating element or an interferometer mirror. These devices are typically interfaced to the processor through device registers and may use interrupts to synchronize with the processor. In order to make the programs easier to understand, it is convenient to introduce a *hardware abstraction layer* (HAL),

Author's address: Martin Schoeberl, Institute of Computer Engineering, Vienna University of Technology, Treitlstr. 3, A-1040 Vienna, Austria. Stephan Korsholm and Anders P. Ravn, Department of Computer Science, Aalborg University, Selma Lagerlöfs vej 300, DK-9220 Aalborg, Denmark, Tomas Kalibera, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907-2107, USA.

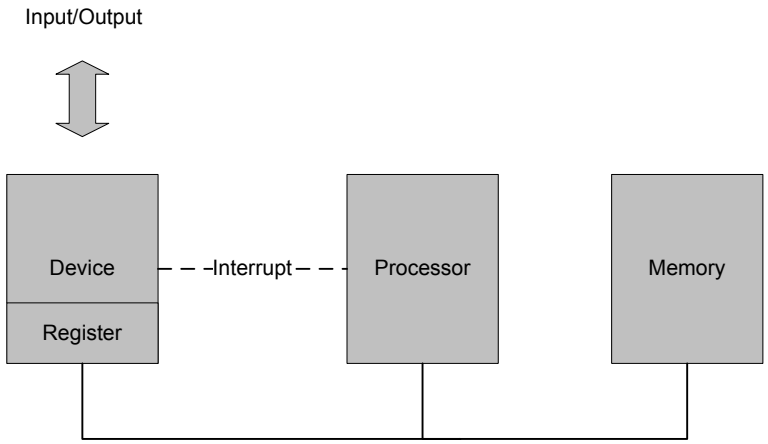

Fig. 1. The hardware: a bus connects a processor to I/O devices and memory, and an interrupt bus connects devices to a processor

where access to device registers and synchronization through interrupts are hidden from conventional program components. A HAL defines an interface in terms of the constructs of the programming language used to develop the application. Thus the challenge is to develop an abstraction that gives efficient access to the hardware while staying within the computational model provided by the programming language.

Our first ideas on a HAL for Java have been published in [Schoeberl et al. 2008] and [Korsholm et al. 2008]. This paper combines the two papers, provides a much wider background of related work, gives two additional experimental implementations, and gives performance measurements that allow an assessment of the efficiency of the implementations. The remainder of this section introduces the concepts of the Java based HAL.

1.1 Hardware Assumptions

The hardware is built up along one or more buses – in small systems typically only one – that connects the processor with memory and device controllers. Device controllers have reserved some part of the address space of a bus for its device registers. They are accessible for the processor as well, either through special I/O instructions or by ordinary instructions when the address space is the same as the one for addressing memory, a so called memory mapped I/O solution. In some cases the device controller will have direct memory access (DMA) as well, for instance for high speed transfer of blocks of data. Thus the basic communication paradigm between a controller and the processor is shared memory through the device registers and/or through DMA. With these facilities only, synchronization has to be done by testing and setting flags, which means that the processor has to engage in some form of busy waiting. This is eliminated by extending the system with an interrupt bus, where the device controllers can generate a signal that will interrupt the normal flow of execution in the processor and direct it to an interrupt handling program. Since communication is through shared data structures, the processor and the controllers need a locking mechanism, such that interrupts can be enabled or disabled by the processor through an interrupt control unit. The typical hardware organization is summarized in Figure 1.

```
typedef struct {
    volatile int data;
    volatile int control;
} parallel_port;
#define PPORT (parallel_port *) 0x10000;

int inval, outval;
parallel_port *mypp;
mypp = PPORT;
...
inval = mypp->data;
mypp->data = outval;
```

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

1.2 A Computational Model

In order to develop a HAL, the device registers and interrupt facilities must be mapped to programming language constructs such that their use corresponds to the computational model underlying them.

A very popular language for developing embedded systems is C. It has a very simple computation model with one thread of execution, starting with main() and evolving through nested subroutine invocations that modify local and global state variables which are weakly typed; i.e. type information is mainly used for memory space allocation and not used to enforce type checking. This gives a very simple mapping for device registers which we introduce through an example.

1.2.1 *An Example in C.* Consider a simple I/O device, e.g. a parallel input/output (PIO) device controlling a set of input and output pins. The PIO uses two registers: the *data register* and the *control register*. Writing to the data register stores the value into an internal latch 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 – 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 hardware with a distinct I/O address space, (e.g. x86), access to device registers is via distinct machine instructions. Those instructions are represented by C functions that take the address as argument.

These simple representations of device registers in C are efficient; but due to the weak typing combined with pointer manipulation they are not very safe.

1.2.2 *An Example in Java.* In an object oriented (OO) language the most natural way to represent an I/O device is as an object – the hardware object. Figure 3 shows a class definition and object instantiation for the simple parallel port. The class ParallelPort corresponds 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 pointer manipula-

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

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

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

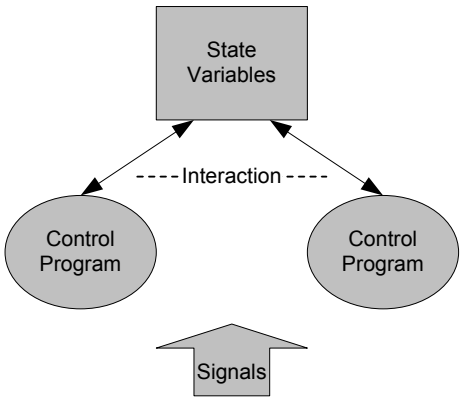

Fig. 4. Computational model: several threads of execution communicate via shared state variables and receive signals.

tion. To provide this convenient representation of I/O devices as objects a JVM internal mechanism is needed to access the device registers via object fields and to *create* the device object and receive a reference to it. We elaborate on the idea of hardware objects in Section 3 and present implementations in Section 6.

1.3 Interrupts

When we consider interrupts, the computational model of C shows its limitations. It is quite easy to associate a C function with an interrupt, so the function is invoked to handle an interrupt; but it may not be that easy to communicate and synchronize correctly with the main program; because there are now conceptually several threads of execution (the main thread and the interrupt controller scheduled handler threads). The solution is well-known; one will use C together with a standard component that provides a multiprogramming kernel with suitable synchronization primitives. Yet, in order to program the kernel, the kernel must use the interrupt facilities, at least if it provides preemptive scheduling. This means that interrupt handlers interact strongly with the kernel such that interfacing to devices becomes an expert task. Yet, conceptually we have arrived at a desired computational model for the programmer shown in Figure 4. The signals are external, asynchronous events that map to interrupts.

A layered implementation of this model with a kernel close to the hardware and ap-

```
public class RS232ReceiveInterruptHandler extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;

    private byte UartRxBuffer[];
    private short UartRxWrPtr;

    ...

    protected void handleInterrupt() {

        synchronized(this) {
            UartRxBuffer[UartRxWrPtr++] = rs232.P0_UART_RX_TX_REG;
            if (UartRxWrPtr >= UartRxBuffer.length) UartRxWrPtr = 0;
        }
        rs232.P0_CLEAR_RX_INT_REG = 0;
        interruptControl.RESET_INT_PENDING_REG = RS232.CLR_UART_RX_INT_PENDING;
    }
}
```

Fig. 5. An example interrupt handler for an RS232 interface. The private object rs232 is a Hardware Object for the concrete device registers. The object interruptControl is an object that interfaces to the interrupt control unit of the processor. The remaining objects define a cyclic byte buffer for receiving bytes from the device. The interrupt handling includes moving a byte from the device register to the buffer and clearing the pending interrupt.

plications on top has been very useful in general purpose programming, where one may even extend the kernel to manage resources and provide protection mechanisms such that applications are safe from one another as for instance when implementing trusted interoperable computing platforms [Group 2008]. Yet there is a price to pay which may make the solution less suitable for embedded systems: Adding new device drivers is an error-prone activity [Chou et al. 2001], and protection mechanisms impose a heavy overhead on context switching when accessing devices.

The alternative we propose is to use Java directly, since it already supports multithreading, and use methods in the special InterruptHandler objects to handle interrupts. The idea is illustrated in Figure 5 and the details, including synchronization and interaction with the interrupt control, are elaborated in Section 4. Implementations are found in Section 6.

1.4 Language Support for a HAL

Already in the 1970s it was recognized that an operating system might not be the optimal solution for special purpose applications. Device access has been integrated into general purpose programming languages. Languages like Concurrent Pascal [Hansen 1977; Ravn 1980] and Modula (Modula-2) [Wirth 1977; 1982] along with a number of similar languages, e.g. UCSD Pascal, supported concurrency and included interfaces to devices. They were meant to eliminate the need for operating systems and were successfully used in a variety of applications. The programming language Ada, which has been dominant in defence and space applications till this day, may be seen as a continuation of these developments. The advent of inexpensive microprocessors from the mid 1980s and on lead to a regression to assembly and C programming. The hardware platforms were small with limited resources and the developers were mostly electronic engineers which viewed them as electronic controllers. Program structure was not considered a major issue in develop-

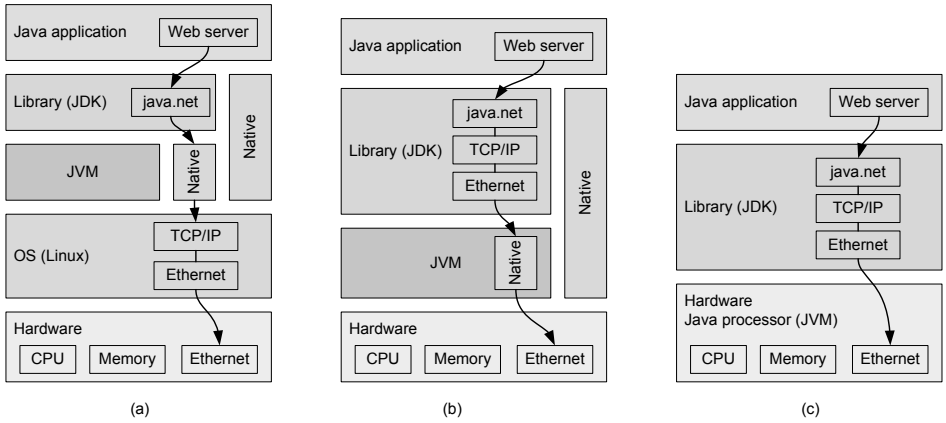

Fig. 6. Configurations for an embedded JVM: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

ment. Nevertheless, the microcomputer has grown, and is now far more powerful than the minicomputer that it replaced. With powerful processors and an abundance of memory, the ambitions for the functionality of embedded systems grow, and programming becomes a major issue, because it may turn out to be a bottleneck in development. A well known solution is to a switch to a higher level language like Java. It results in fewer lines of code and more structured concepts. In order to have a conceptually coherent programming platform this should ideally include the HAL.

1.5 Mapping Between Java and the Hardware

When we study interfacing from Java to the hardware, we do not require language extensions. The Java concepts of packages, classes and synchronized objects turn out to be powerful enough to formulate the desired abstractions. The mapping is done at the level of the Java Virtual Machine (JVM). The JVM already provides typical OS functions handling:

—Address space allocation

—Thread management

—Inter-process communication

These parts need to be modified so they cater for interfaces to the hardware.

Yet, the architectures of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 6 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 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 the thread scheduling and the low-level access to the hardware. In that case the network stack can be written entirely in Java. JNode[1] is an approach to implement the OS entirely in Java. This solution has become popular even in server applications.[2]

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

[2] BEA System offers the JVM LiquidVM that includes basic OS functions and does not need a guest OS.

Sub-figure (c) shows an embedded solution where the JVM is part of the hardware layer. That means it is implemented in a Java processor. With this solution the native layer can be completely avoided and all code (application and system code) is entirely written in Java.

To motivate the proposed HAL Figure 6 also shows the data and control flow from the application down to the hardware. The example consists of a web server and an Internet connection via Ethernet. In case (a) the application web server talks with java.net in the Java library. The flow goes down via a native interface to the TCP/IP implementation and the Ethernet device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the Ethernet device driver are now part of the Java library. In (c) the JVM is part of the hardware layer and a direct access from the Ethernet driver to the Ethernet hardware is mandatory. With our proposed HAL the native interface within the JVM in (b) almost disappears. Note how the network stack moves up from the OS layer to the Java library. Version (c) shows a pure Java implementation of the whole network stack. Version (b) and (c) can benefit from hardware objects and interrupt handler in Java as access to the Ethernet device is required from Java source code. In Section 7 we show a simple web server application implemented completely in Java as evaluation of our approach.

1.6 Contributions

The key contribution of this paper is a proposal for a HAL for Java, such that it can run on the bare metal while still being a *safe language*. This idea is investigated in quite a number of places which are discussed in the related work section where we comment on our initial ideas as well. In summary, the proposal gives an interface to hardware that has the following benefits:

Object-oriented. An object representing a device is the most natural integration into an OO language, and a method invocation to a synchronized object is a direct representation of an interrupt.

Safe. The safety of Java is not compromised. Hardware objects map object fields to the device registers. With a correct class that represents the I/O device, access to the low-level device is safe. Hardware objects can only be created by a Factory residing in a special package.

Efficient. Device register access is performed by single bytecodes getfield and putfield. We can avoid expensive native calls to C. The handlers are first level handlers; there is no delay through event queues.

The proposed Java HAL would not be useful if it had to be modified for each particular kind of JVM, thus a second contribution of this paper is a number of prototype implementations illustrating the architectures presented in Figure 6: implementations in Kaffe [Wilkinson 1996] and OVM [Armbruster et al. 2007] represent the architecture with an OS in (sub-figure (a)), the implementation in SimpleRTJ [RTJ Computing 2000] represent the bare metal solution in (sub-figure (b)), and the implementation in JOP [Schoeberl 2008] represents the Java processor solution (sub-figure (c)).

Finally we must not forget efficiency, thus the paper ends with some performance measurements that indicate that the HAL layer is generally as efficient as native calls to C code external to the JVM.

2. RELATED WORK

An excellent overview of historical solutions to access hardware devices from and implement interrupt handlers in high-level languages, including C, is presented in Chapter 15 of [Burns and Wellings 2001]. The solution to device register access in Modula-1 (Ch. 15.3) is very much like C; however the constructs are safer, because they are encapsulated in modules. Interrupt handlers are represented by threads that block to wait for the interrupt. In Ada (Ch 15.4) the representation of individual fields in registers can be described precisely using *representation* classes, while the corresponding structure is bound to a location using the Address attribute. An interrupt is represented in the current version of Ada by a protected procedure, although initially represented (Ada 83) by task entry calls.

2.1 The Real-time Specification for Java

The Real-Time Specification for Java (RTSJ) [Bollella et al. 2000] defines a JVM extension which allows better timeliness control compared to a standard JVM. The core features are: fixed priority scheduling, monitors which prevent priority inversion, scoped memory for objects with limited lifetime, immortal memory for objects that are never finalized, and asynchronous events with CPU time consumption control.

The RTSJ also defines an API for direct access to physical memory, including hardware I/O registers. Essentially, one has to use RawMemoryAccess at the level of primitive data types. 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. A type safe layer with support for representing individual registers can however be implemented on top of the RTSJ API.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. It however neither specifies an API for interrupt control, nor semantics of the handlers. Any interrupt handling application thus has to rely on some proprietary API and proprietary restrictions of event handler semantics, which violate the RTSJ specification. Second level interrupt handling can be implemented within the RTSJ with an AsyncEvent that can be bound to a *happening*. The happening is a string constant that can represent an interrupt, but the meaning is implementation dependent. An AsyncEventHandler or BoundAsyncEventHandler can be added as the handler for the event. An AsyncEventHandler can be added via POSIXSignalHandler to handle POSIX signals. An interrupt handler, written in C, can then use one of the two available POSIX user signals.

We are not aware of any RTSJ implementation that implements RawMemoryAccess and AsyncEvent with support for low-level device access and interrupt handling. An interesting idea to explore would be to define and implement a two-level scheduler for the RTSJ. It should provide the first level interrupt handling for asynchronous events bound to interrupts and delegate other asynchronous events to an underlying second level scheduler, which could be the standard fixed priority preemptive scheduler. This would be a fully RTSJ compliant implementation of our proposal.

2.2 Hardware Interface in JVMs

The aJile Java processor [aJile 2000] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix

suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this lock. When the thread is unblocked and holds the lock, effectively all interrupts are disabled.

Komodo [Kreuzinger et al. 2003] is a multithreaded Java processor targeting real-time systems. On top of the multiprocessing pipeline the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. That thread is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete thread state for the interrupt service thread in the hardware. In the case of Komodo that state consists of an instruction window and a stack register stack. Devices are represented by Komodo specific I/O classes.

Muvium [Caska] is an ahead-of-time compiling JVM solution for very resource constrained microcontrollers (Microchip PIC). Muvium uses an Abstract Peripheral Toolkit (APT) to represent I/O devices. APT is based on an event driven model for the interaction with the external world. Device interrupts and periodic activations are represented by events. Internally, events are mapped to threads with priority and a preemptive scheduler. APT contains a large collection of classes to represent devices common in embedded systems.

2.3 Java Operating Systems

The JX Operating System [Felser et al. 2002] is a microkernel system written mostly in Java. The system is composed of components which run in *domains*, each domain having its own garbage collector, threads, and a scheduler. There is one global preemptive scheduler which schedules the domain schedulers, which can be both preemptive and non-preemptive. Inter-domain communication is only possible though communication channels exported by services. Low level access to the physical memory, memory mapped device registers and I/O ports is provided by the core ("zero") domain's services, implemented in C. At Java level, ports and memory areas are represented by objects, and registers by methods of these objects. Memory is read and written by access methods of Memory objects. Higher layers of Java interfaces provide type safe access to the registers; the low level access is not type safe.

Interrupt handlers in JX are written in Java and are run through portals – they can reside in any domain. Interrupt handlers cannot interrupt the garbage collector (the GC disables interrupts), run with CPU interrupts disabled, must not block, and can only allocate a restricted amount of memory from a reserved per domain heap. Execution time of interrupt handlers can be monitored: on a deadline violation the handler is aborted and the interrupt source disabled. The first level handlers can unblock a waiting second level thread, either directly, or via setting a state of a AtomicVariable synchronization primitive.

The Java New Operating System Design Effort (JNode[4]) [Lohmeier 2005] is an OS written in Java where the JVM serves as the OS. Drivers are written entirely in Java. I/O access is performed via native function calls. The first level interrupt handler, written in assembler, unblocks a Java interrupt thread. From that thread the device driver level interrupt

[3] An template can be found at `http://practicalembeddedjava.com/tutorials/aJileISR.html`
[4] `http://jnode.org/`

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Fig. 7. The marker class for hardware objects

handler is invoked with interrupts disabled. Some device drivers implement handleInterrupt(int irq) synchronized and use the driver object to signal the upper layer with notifyAll(). During garbage collection all threads are stopped including the interrupt threads.

The Squawk VM [Simon et al. 2006], now available open-source[5], is a platform for wireless sensors. Squawk is mostly written in Java and runs without an OS. Device drivers are written in Java and use a form of peek and poke interface to access the device's memory. Interrupt handling is supported by a device driver thread that waits for an event from the VM. The first level handler, written in assembler, disables the interrupt and notifies the JVM. On a rescheduling point the JVM resumes the device driver thread. That thread has to re-enable the interrupt. The interrupt latency depends on the rescheduling point and on the activity of the garbage collector. For a single device driver thread an average case latency of 0.1 ms is reported. For a realistic workload with an active garbage collector a worst-case latency of 13 ms has been observed.

Singularity [Hunt et al. 2005] is a research OS based on a runtime managed language (an extension of C#) to build a software platform with the main goal to be dependable. A small HAL (IoPorts, IoDma, IoIrq, and IoMemory) provides access to PC hardware. C# style attributes (similar to Java annotations) in fields are used to define the mapping of class fields to I/O ports and memory addresses.

2.4 Summary

The distinctive feature of our proposal is that it allows mapping a device into the OO address space and providing, if desired, access methods for individual fields, such that it lifts the facilities of Ada into the object oriented world of Java. Furthermore, we provide a path to implement true first level interrupt handlers in Java. Those two concepts allow implementation of a Java based system with almost no code written in assembler or C (except processor initialization).

3. DEVICE ACCESS

Hardware objects map object fields to the device registers. Therefore, the field access with bytecodes putfield and getfield accesses the device registers. With a correct class that represents an I/O device, access to the low-level device is safe – it is not possible to read or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

All hardware classes have to extend the abstract class HardwareObject (see Figure 7). This empty class serves as type marker. It is needed by some implementations to distinguish between plain objects and hardware objects for the field access. The package visible constructor disallows creation of hardware objects by the application code that resides in a different package.

Figure 8 shows a class representing a serial port with a status register and a data register.

[5] https://squawk.dev.java.net/

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
    public volatile int data;

    public void init(int baudRate) {...}
    public boolean rxFull() {...}
    public boolean txEmpty() {...}
}
```

Fig. 8. A serial port class with device methods

The status port contains flags for receive register full and transmit register empty; the data register represents the receive and transmit buffer. Furthermore, we define device specific constants (bit masks for the status register) in the class for the serial port. All fields represent device registers that can change due to activity of the hardware device. Therefore, they must be declared volatile.

In this example we have also added some convenient methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, i.e., mixing device registers with words in the heap. That means we cannot implement a complete device driver within a hardware object. Instead, a complete device driver owns a number of private hardware objects along with data structures for buffering, and it then defines interrupt handlers and methods for accessing its state from application processes. For simple operations, such as initialization of the device, methods in hardware objects can be useful.

3.1 Access Control

Each device is usually represented by exactly one hardware object (see Section 5.1). However, there might be use cases where this restriction should be relaxed. Consider a device where some registers should only be accessed by system code and some other by application code. In JOP we have such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler – a JVM internal service. However, access to the counter can be useful for the application code. And access to the watchdog register is mandatory at the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it might be useful to represent one hardware device by two *different* classes – one for system code and one for application code. We can protect system level registers by private fields in the hardware object for the application. Figure 9 shows the two class definitions that represent the same hardware device for system and application code. Note how we changed the access to the timer interrupt register to private for the application hardware object.

Another option, shown in class AppGetterSetter, is to declare all fields private for the application object and use setter and getter methods. Getter and setter methods add just

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}

public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Fig. 9. System and application classes, one with visibility protection and one with setter and getter methods, for a single hardware device

another abstraction on top of hardware objects. The methods use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

3.2 Using Hardware Objects

Usage of hardware objects is straightforward. 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 10 shows an example of the client code. The example is the *Hello World* program using low-level access to the serial port via a hardware object.

Creation of hardware objects is a more complex and described in Section 5. Furthermore, it is JVM specific and Section 6 provides the description of implementations in four different JVMs.

3.3 Hardware Arrays

For devices that use DMA (e.g., video frame buffer, disk and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking inherent to the JVM specification. Hardware arrays can be *used* by the JVM to *implement* higher-level abstractions from the RTSJ such as RawMemory or scoped memory.

```
import com.jopdesign.io.*;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

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

Fig. 10. The Hello World example with low-level device access via a hardware object

3.4 Garbage Collection

The interaction between the garbage collector (GC) and hardware objects needs to be crafted into the JVM. We do not want to *collect* hardware objects. In the general case the hardware object should not be scanned for references.[6] The second property is fulfilled when only primitive types are used in the class definition for hardware objects – the GC scans only reference fields. For the first property we have to *mark* the object to be skipped by the GC. The type inheritance from HardwareObject can be used as the marker.

3.5 Summary

Hardware Objects are easy to use for a programmer, and the corresponding definitions are comparatively easy to define for a hardware designer or manufacturer. For a standardized HAL architecture, we propose factory patterns outlined in Section 5. Implementation has a few subtle points which are discussed in Section 6.

4. INTERRUPT HANDLING

In a Java context, an interrupt can be serviced by an interrupt service routine (ISR) in three different ways:

Handler. The interrupt is a method call initiated by the device processor. This abstraction is usually supported in hardware by the processor and called a first level handler.

Thread. A thread is unblocked from a wait-state by the interrupt. Direct hardware support for such signal operations are not common, because it requires hardware support for thread handling in general. It can, however, be implemented by a scheduler, or imple-

[6]If a hardware coprocessor, represented by a hardware object, itself manipulates an object on the heap and holds the only reference to that object it has to be scanned by the GC.

ISR	Context switches	Priorities
Handler	2	Hardware
Thread	3–4	Software
Event	3–4	Software

Table I. Scheduling properties of different ISR strategies

mented by the *Handler* method mentioned above. The thread is then a second level handler.

Event. The interrupt is represented by an asynchronous notification directed to a thread as for the previous solution. This is also called deferred interrupt handling.

An overview of the scheduling properties of these three approaches is given in Table I. The ISR Handler approach needs only two context switches and the priority is set by the hardware. All three approaches are modeled as a sporadic or aperiodic event for the scheduling analysis.

The ISR Thread and the ISR Event approach are quite similar with respect to scheduling and context switches. Both are scheduled at software priorities. The initial first level handler, running at hardware priority, either unblocks the thread for release or fires the event for the event handler. The first level handler will also notify the scheduler. In the best case three context switches are necessary: one into the first level handler, one to the ISR thread or event handler, and one back to the interrupted thread. If the ISR thread or handler has a lower priority than the interrupted thread an additional context switch (from the first level handler back to the interrupted thread) is performed.

It has to be noted that the ISR thread abstraction can run at hardware priority, similar to the handler approach, when supported by the hardware (e.g. the interrupt service thread in Komodo [Kreuzinger et al. 2003]). However, as standard processors support only the handler model directly in hardware, we consider the ISR Thread approach as deferred handling at software priority. In the following we describe the consequences of choosing one approach over the other.

4.1 ISR Handler

To support direct handling of interrupts in Java, the JVM must be prepared to be interrupted. In an interpreting JVM an initial handler will reenter the JVM interpreter to execute the Java handler. A compiling JVM or a Java processor can directly invoke a Java method as response to the interrupt. A compiled Java method can be registered directly in the ISR dispatch table.

If an internal scheduler is used (also called *green threads*) the JVM will need some refactoring in order to support asynchronous method invocation. JVMs usually control the rescheduling at the JVM level to provide a lightweight protection of JVM internal data structures. Those preemption points are called pollchecks or GC preemption points. In fact those preemption points resemble cooperative scheduling at the JVM level and use priority for synchronization. This is an approach that works only for uniprocessor systems.

In any case there might be critical sections in the JVM where reentry cannot be allowed. To solve it, the JVM must disable interrupts around critical non-reenterable sections. The more fine grained this disabling of interrupts can be done the more responsive to interrupts the system will be, and the more the Java interrupt handling resembles legacy interrupt

handling.

4.2 ISR Thread

In its simplest form, handing over the interrupt to a Java thread (as opposed to handling it directly), will look like the following:

```
for (;;) {
    waitForInterrupt(interruptID);
    handleInterrupt();
}
```

The call to waitForInterrupt blocks the calling thread until an interrupt occurs. As part of its initialization the JVM will populate the interrupt vector table with very simple handlers. The handler will just release the threads blocked in the call to waitForInterrupt. The advantage of this approach is that the handling of the interrupt is done in the context of a normal Java thread and scheduled as any other thread running on the system. It is a simple strategy that does not require the introduction of concepts not inherently part of the Java language. The drawback is that there will be a delay from the actual occurrence of the interrupt until the thread gets scheduled. Additionally the meaning of interrupt priorities, levels and masks as used by the hardware may not map directly to scheduling parameters supported by the JVM scheduler.

4.3 ISR Event

Extending on the idea from above, the occurrence of an interrupt can be expressed as an event that fires and releases an associated schedulable object (handler). Once released, the handler will be scheduled by the JVM scheduler according to the release parameters. This is the form suggested by the RTSJ to handle interrupts. The advantages and disadvantages are the same as for the ISR Thread approach.

In the following we focus on the *Handler* approach, because it allows programming the other paradigms within Java.

4.4 Hardware Properties

We assume interrupt hardware as it is found in most computer architectures: interrupts have a fixed priority associated with them – they are set with a *solder iron*. Furthermore, interrupts can be globally disabled. On most systems the first level handler is called with interrupts globally disabled. To allow nested interrupts – being able to interrupt the handler by a higher priority interrupt as in preemptive scheduling – the handler has to enable interrupts again. However, to avoid priority inversion between handlers only interrupts with a higher priority will be enabled; either by setting an interrupt level or setting an interrupt mask.

Software threads are scheduled by the timer interrupt and usually have a lower priority than interrupt handlers (the timer interrupt has the lowest priority of all interrupts). Therefore, an interrupt handler is never preempted by a software thread.

To ensure mutual exclusion between an interrupt handler and a software thread the software thread has to protect the critical sections by disabling interrupts: either disable all interrupts or selectively disable interrupts. Again, to avoid priority inversion, only interrupts of a higher priority than the interrupt that shares the data with the software thread can be enabled. This mechanism is in effect the priority ceiling emulation protocol [Sha et al.

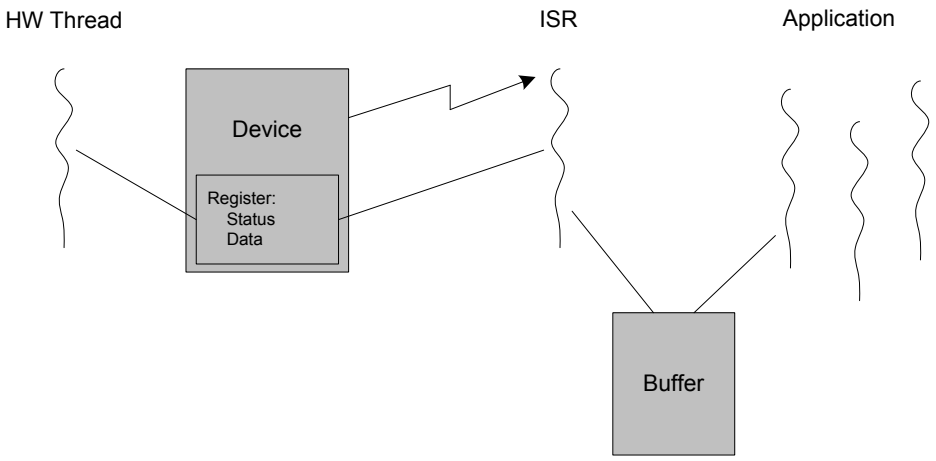

Fig. 11. Threads and shared data

1990], sometimes called immediate ceiling protocol, to avoid priority inversion. It has the virtue that it eliminates the need for explicit locks on shared objects.

Note that ensuring mutual exclusion with interrupt disabling works only in a uniprocessor setting. A simple solution for multiprocessors is to run the interrupt handler and associated software threads on the same processor core.

4.4.1 *Mechanics of First Level Handlers.* When an I/O device puts a signal on an interrupt request line, the interrupt controller notifies the processor. The processor stops execution of the instructions of the current thread. A partial thread context (program counter and processor status register) is saved. Then the ISR is looked up in the interrupt vector table and a jump is performed to the first instruction of the ISR.

The handler usually saves additional thread context (e.g. the register file). It is also possible to switch to a new stack area. This is important for embedded systems without virtual memory where the stack sizes for all threads need to be determined at link time.

4.5 Synchronization

Java supports synchronization between Java threads through the synchronized keyword, either as a means of synchronizing access to a block of statements or to an entire method. In the general case this existing synchronization support is not sufficient to synchronize between interrupt handlers and threads.

Figure 11 shows the interacting active processes and the shared data in a scenario involving the handling of an interrupt. Conceptually three processes interact: (1) a hardware (or device) thread representing the device activity, (2) the ISR (implemented using one of the three strategies described above), and (3) the application process. These three threads share two types of data:

Device data. The hardware thread and ISR share access to the device registers of the device signaling the interrupt

Application data. The ISR and application share access to e.g., a buffer conveying information about the interrupt to the application

ISR	Object lock	Shared object lock	wait	notify
Handler	interrupt disable	no	no	ok
Thread	standard monitor	yes	no	ok
Event	standard monitor	yes	no	ok

Table II. Synchronization properties of different ISR strategies

Regardless of which of these approaches is used to handle interrupts in Java, synchronization around access to device data must be handled in an ad hoc way. There is in general no guarantee that the device thread has not changed the data in the I/O register. But in general, if the ISR can be run to completion within the minimum inter arrival time of the interrupt the content of the device registers can be trusted.

Table II gives an overview of the synchronization properties of the three approaches between the ISR and the application threads. As ISR Thread and Event handler run as software threads standard synchronization with object locks can be used. In any case an ISR shall never block (wait is forbidden), but is allowed to notify blocked applications or device driver threads.

When using the ISR Handler approach, the handler is no longer scheduled by the normal Java scheduler, but by the hardware. While the handler is running all other executable elements are suspended, including the scheduler. This means that the ISR cannot be suspended, must not block, or must not wait on a language level synchronization mechanism. Rather the ISR must run to completion in order not to freeze the system. This means that when the handler runs, it is guaranteed that the application will not get scheduled. It follows that the handler can access data shared with the application without synchronizing with the application. As the access to the shared data by the interrupt handler is not explicitly protected by a synchronized method or block, the shared data needs to be declared volatile.

On the other hand the application must be synchronized with the ISR, but the ISR may get scheduled at any point. To ensure mutual exclusion we redefine the semantics of the lock associated with an InterruptHandler object: acquisition of the lock disables all interrupts of the same and lower priority; release of the lock enables the interrupts again. This algorithm ensures that the software thread cannot be interrupted by the interrupt handler when accessing shared data. That behavior can be approximated by globally disabling interrupts on critical sections. However, that results also in blocking of higher priority interrupts.

4.6 Using the Interrupt Handler

Figure 12 shows an example of an interrupt handler for the serial port receiver interrupt. The method handle() is the interrupt handler method and needs no synchronization as it cannot be interrupted by a software thread. However, the shared data needs to be declared volatile as it is changed by the device diver thread. Method read() is invoked by the device driver thread and the shared data is protected by the InterrupHandler lock. The serial port interrupt handler uses the hardware object SerialPort to access the device.

```
public class SerialHandler extends InterruptHandler {

    // A hardware object represents the serial device
    private SerialPort sp;

    final static int BUF_SIZE = 32;
    private volatile byte buffer[];
    private volatile int wrPtr, rdPtr;

    public SerialHandler(SerialPort sp) {
        this.sp = sp;
        buffer = new byte[BUF_SIZE];
        wrPtr = rdPtr = 0;
    }

    // This method is scheduled by the hardware
    public void handle() {
        byte val = (byte) sp.data;
        // check for buffer full
        if ((wrPtr+1)%BUF_SIZE!=rdPtr) {
            buffer[wrPtr++] = val;
        }
        if (wrPtr>=BUF_SIZE) wrPtr=0;
        // enable interrupts again
        enableInterrupt();
    }

    // This method is invoked by the driver thread
    synchronized public int read() {
        if (rdPtr!=wrPtr) {
            int val = ((int) buffer[rdPtr++]) & 0xff;
            if (rdPtr>=BUF_SIZE) rdPtr=0;
            return val;
        } else {
            return -1;        // empty buffer
        }
    }
}
```

Fig. 12. An example interrupt handler for the serial port receive interrupt

4.7 Garbage Collection

When using the ISR Handler approach, it is apparent that the ISR must not block waiting for a GC to finish. It is not feasible to let interrupt handlers start a lengthy stop-the-world collection. Both facts affect the inter-operability of interrupt handlers with a GC.

4.7.1 *Stop-the-world GC.* Using this strategy the entire heap is collected at once and the collection is not interleaved with execution. The collector can safely assume that data required to perform the collection will not change during the collection, and an interrupt handler shall not change data used by the GC to complete the collection. In the general case, this means that the interrupt handler is not allowed to create new objects, or change the graph of live objects.

4.7.2 *Incremental GC.* The heap is collected in small incremental steps. Write barriers in the mutator threads and non-preemption sections in the GC thread synchronize the view of the object graph between the mutator threads and the GC thread.

With incremental collection it is possible to allow object allocation and changing references inside an interrupt handler (as it is allowed in any normal thread). With a real-time GC the maximum blocking time due to GC synchronization with the mutator threads must be known.

4.7.3 *Moving Objects.* Interruption of the GC during an object move can result in access to a stale copy of the object inside the handler. A possible solution to this problem is to allow for pinning of objects reachable by the handler (similar to immortal memory in the RTSJ). Concurrent collectors have to solve this issue anyway for the concurrent threads. The simplest approach is to disable interrupt handling during the object copy. As this operation can be quite long for large arrays, several approaches to split the array into smaller chunks have been proposed [Siebert 2002] and [Bacon et al. 2003]. A Java processor may support incremental array copying with redirection of the access to the correct part of the array [Schoeberl and Puffitsch 2008].

4.8 Summary

As shown, interrupt handlers are easy to use for a programmer that knows the underlying hardware paradigm, and the definitions are comparatively easy to define for a hardware designer or manufacturer, for instance using the patterns outlined in Section 5. Implementation has quite a number of subtle points which are discussed in Section 6.

5. DEVICE CONFIGURATION DEFINITIONS

An important issue for the concept of hardware objects and Java interrupt handlers is a safe abstraction of device configurations. The definition of hardware objects and factories to create such objects should be provided by the board vendors. This configuration is composed in Java packages and only the hardware objects and the Factory methods are visible.

The examples in the following section are taken from the implementation on JOP as this system comes with many different I/O configurations. The implementations on Kaffe and OVM that run on a PC use the same abstraction.

5.1 Hardware Object Creation

The idea to represent each I/O device by a single object or array is straightforward, the remaining question is: How are those objects created? An object that represents an I/O device is a typical Singleton [Gamma et al. 1994]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

The board manufacturer provides the classes for the hardware objects and the configuration class for the board. The board configuration class provides the Factory [Gamma et al. 1994] 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 (the application runs

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

public class IOSystem {

    // some JVM mechanism to create the hardware objects
    private static ParallelPort pp = jvmPPCreate();
    private static SerialPort sp = jvmSPCreate(0);
    private static SerialPort gps = jvmSPCreate(1);

    public static ParallelPort getParallelPort() {
        return pp;
    }

    public static SerialPort getSerialPort() {..}
    public static SerialPort getGpsPort() {..}
}
```

Fig. 13. A Factory with static methods for Singleton hardware objects

on a single board). The Singleton property of the configuration is enforced by a class that contains only static methods. Figure 13 shows an example for such a class. The class IOSystem represents a system with three devices: a parallel port as discussed before to interact with the environment and two serial ports one for program download and one for an interface to a GPS receiver.

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 (or additional) IOSystem class that lists all devices. A more elegant solution is proposed in the next section.

5.2 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 14 shows an example of a base Factory and a derived Factory. Note how getBaseFactory() 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 base Factory represents the minimum configuration with a single serial port for communication (mapped to System.in and System.out) represented by a SerialPort. The derived configuration ExtendedBoard contains an additional serial port for a GPS receiver and a parallel port for external control.

Furthermore, we show in Figure 14 a different way to incorporate the JVM mechanism into the Factory: we define well known constants (the memory addresses of the devices) in the Factory and let the native function jvmHWOCreate() return the correct I/O device type.

Figure 15 gives a summary example of hardware object classes and the corresponding Factory classes as an UML class diagram. The figure shows that all I/O classes subclass

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}

public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = (ParallelPort) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort() { return parallel; }
}
```

Fig. 14. A base class of a hardware object Factory and a Factory subclass

the abstract class HardwareObject.

5.3 Interrupt Handler Registration

We provide a base interrupt handling API that can be used both for non-RTSJ and RTSJ interrupt handling. The skeleton of a handler is shown in Figure 16. The handle() method contains the device serving code. Interrupt control operations that have to be invoked before serving the device (i.e. interrupt masking and acknowledging) and after serving the device (i.e. interrupt re-enabling) are hidden in the run() method of the base InterruptHandler, which is invoked when the interrupt occurs – see Figure 17.

The base implementation of InterruptHandler also provides methods for enabling and disabling a particular interrupt or all local CPU interrupts and a special monitor implementation for synchronization between an interrupt handler thread and a single application thread. Moreover, it provides methods for non-RTSJ registering and deregistering the handler with the hardware interrupt source.

The registration of the interrupt handler in RTSJ requires more steps (see Figure 18).

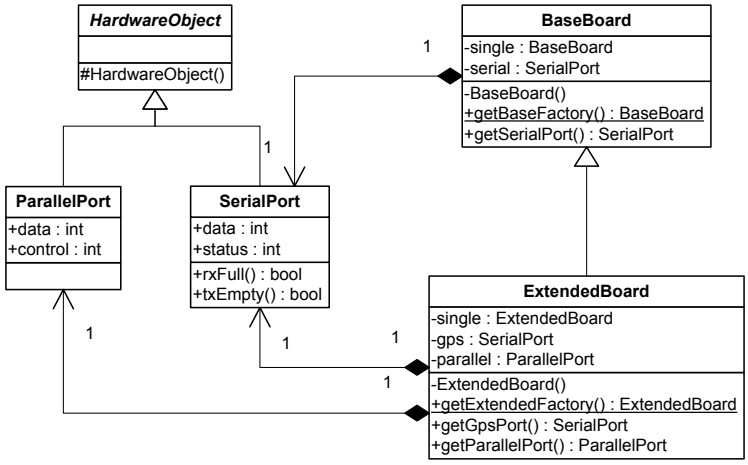

Fig. 15. Hardware object classes and board Factory classes

```
private class SerialInterruptHandler extends InterruptHandler {

    public SerialInterruptHandler() {
        super(SERIAL_INTERRUPT_INDEX);
    }

    public void handle() {
        // first level handling code
    }
}
```

Fig. 16. Skeleton of a first level interrupt handler

The InterruptHandler instance serves as the RTSJ logic for a (bound) asynchronous event handler, which is added as a handler to an asynchronous event, which is then bound to the interrupt source.

6. IMPLEMENTATION

To verify the proposed Java HAL, we have implemented the core concepts on four different JVMs.[7] Table III classifies the four execution environments according to two important properties (1) whether they run on the bare metal or on top of an OS and (2) whether Java code is interpreted or executed natively. Therefore, we cover the whole implementation spectrum with our four implementations. The reason for including systems running on top of an OS is that even though the suggested Java HAL is intended for systems running bare bone, most existing JVMs still require an OS, and in order for them to migrate incrementally to run directly on the hardware they can benefit from supporting a Java HAL.

In the direct implementation a JVM without an OS is extended with I/O functionality. The indirect implementation represents an abstraction mismatch – we actually re-map the concepts. Related to Figure 6 in the introduction, OVM and Kaffe represent configuration

[7]On JOP the implementation of the Java HAL is already in use in production code.

```
abstract public class InterruptHandler implements Runnable {
    ...

    public InterruptHandler(int index) { ... };

    protected void startInterrupt() { ... };
    protected void endInterrupt() { ... };

    protected void disableInterrupt() { ... };
    protected void enableInterrupt() { ... };
    protected void disableLocalCPUInterrupts() { ... };
    protected void enableLocalCPUInterrupts() { ... };

    public void register() { ... };
    public void unregister() { ... };

    abstract public void handle() { ... };

    public void run() {
        startInterrupt();
        handle();
        endInterrupt();
    }
}
```

Fig. 17. Base class for the interrupt handlers

```
ih = new SerialInterruptHandler(); // logic of new BAEH

serialFirstLevelEvent = new AsyncEvent();
serialFirstLevelEvent.addHandler(
    new BoundAsyncEventHandler( null, null, null, null, null, false, ih )
    );

serialFirstLevelEvent.bindTo("INT4");
```

Fig. 18. Creation and registration of an RTSJ interrupt handler

	Direct (no OS)	Indirect (OS)
Interpreted	SimpleRTJ	Kaffe VM
Native	JOP	OVM

Table III. Embedded Java Architectures

(a), SimpleRTJ configuration (b), and JOP configuration (c).

The SimpleRTJ JVM [RTJ Computing 2000] is a small interpreting JVM that does not require an OS. JOP [Schoeberl 2005; 2008] is a Java processor executing Java bytecodes directly in hardware. The Kaffe JVM [Wilkinson 1996] is a complete, full featured JVM supporting both interpretation and JIT compilation; in our experiments with Kaffe we have used interpretative execution only. The OVM JVM [Armbruster et al. 2007] is an execution environment for Java that supports the compilation of Java bytecodes into the C language, and via a C compiler into native machine instructions for the target hardware. Hardware

```
public final class SerialPort extends HardwareObject {
    // LSR (Line Status Register)
    public volatile int status;
    // Data register
    public volatile int data;
    ...
}
```

Fig. 19. A simple hardware object

objects have also been implemented in the research JVM CACAO [Krall and Grafl 1997; Schoeberl et al. 2008].

In the following section we provide the different implementation approaches that are necessary for the very different JVMs. Implementing hardware objects was straight forward for most JVMs; it took about one day to implement them in JOP. In Kaffe, after familiarizing us with the structure of the JVM, it took about half a day of pair programming.

Interrupt handling in Java is straight forward in a JVM not running on top of an OS (JOP and SimpleRTJ). Kaffe and OVM both run under vanilla Linux or the real-time version Xenomai Linux [Xenomai developers 2008]. Both versions use a distinct user/kernel mode and it is not possible to register a user level method as interrupt handler. Therefore, we used threads at different levels to simulate the Java handler approach. The result is that the actual Java handler is the 3rd or even 4th level handler. This solution introduces quite a lot of overheads due to the many context switches. However, it is intended to provide a stepping stone to allow device drivers in Java; the goal is a real-time JVM that runs on the bare hardware.

In this section we provide more implementation details than usual to help other JVM developers to add a HAL to their JVM. The techniques used for the JVMs can probably not be used directly. However, the solutions (or sometimes work-around) presented here should give enough insight to guide other JVM developers.

6.1 SimpleRTJ

The SimpleRTJ JVM is a small, simple and portable JVM. We have ported it to run on the bare metal of a small 16 bit microcontroller. We have successfully implemented the support for hardware objects in the SimpleRTJ JVM. For interrupt handling we use the ISR Handler approach as described in Section 4. Adding support for hardware objects was straight forward, but adding support for interrupt handling required some more work.

6.1.1 *Hardware Objects.* Given an instance of a hardware object as define in Figure 19 it must be possible to calculate the base address of the I/O port range, the offset to the actual I/O port, and the width of the port at runtime.

We have chosen to store the base address of the I/O port range in a field in the common super class for all hardware objects (HardwareObject). The hardware object Factory passes the platform specific and device specific base address to the constructor when creating instances of hardware objects (see Figure 20).

In the implementation of the put/getfield bytecodes the base address is retrieved from the object instance. The I/O port offset is calculated from the offset of the field being accessed, in the example in Figure 19 status has an offset of 0, whereas data has an offset of 4. The

```
SerialPort createSerialPort(int baseAddress ) {
    SerialPort sp = new SerialPort(baseAddress);
    return sp;
}
```

Fig. 20. Creating a simple hardware object

width of the field being accessed is the same as the width of the field type. Using these values the SimpleRTJ JVM is able to access the device register for either read or write.

6.1.2 *Interrupt Handler.* The SimpleRTJ JVM uses a simple stop-the-world garbage collection scheme. This means that within handlers, we prohibited use of the new keyword and writing references to the heap. These restrictions can be enforced at runtime by throwing a pre-allocated exception or at class loading by an analysis of the handler method. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 4.7.3.

The SimpleRTJ JVM implements thread scheduling within the JVM. This means that it had to be refactored to allow for reentering the JVM from inside the first level interrupt handler. We got rid of all global state (all global variables) used by the JVM and instead allocate shared data on the C stack. For all parts of the JVM to still be able to access the shared data we pass around a single pointer to that data. In fact we start a new JVM for the interrupt handler.

The SimpleRTJ JVM contains support for a skimmed down version of the RTSJ style interrupt handling facilities using the AsyncEvent and AsyncEventHandler classes. Using the javax.events package supplied with the JVM, a server thread can be started that waits for events to occur. This server thread runs at highest priority. The SimpleRTJ JVM reschedule points are between the executions of the bytecodes. Before the execution of each bytecode the JVM checks if a new event has been signaled. If so, the server thread is scheduled immediately and released to handle the event. To achieve interrupt handling through the ISR Handler approach we force a reentry of the JVM from inside the first level interrupt handler by calling the main interpreter loop. Prior to this we have marked that an event is indeed pending, resulting in the server thread being scheduled immediately. To avoid interference with the GC we switch the heap and stack with a new temporary (small) Java heap and a new temporary (small) Java stack. Currently we use 512 bytes for each of these items, which have proven sufficient for running non-trivial interrupt handlers so far.

The major part of the work was making the JVM reentrant. The effort will vary from one JVM implementation to another, but since global state is a bad idea in any case, JVMs of high quality should use very little global state. Using these changes we have experimented with handling the serial port receive interrupt.

6.2 JOP

JOP is a Java processor intended for hard real-time systems [Schoeberl 2005; 2008]. All architectural features have been carefully designed to be time-predictable with minimal impact on average case performance. We have implemented the proposed HAL in the JVM for JOP. No changes inside the JVM (the microcode in JOP) where necessary. Only the creation of the hardware objects needs a JOP specific Factory.

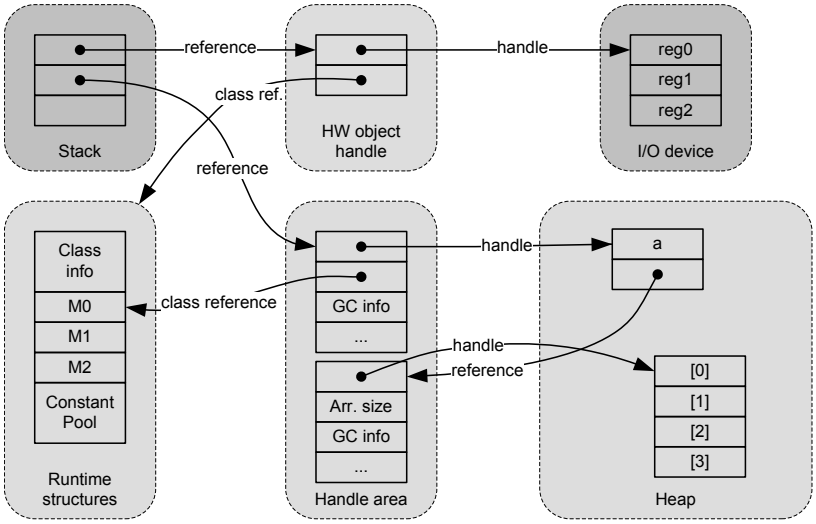

Fig. 21. Memory layout of the JOP JVM

6.2.1 *Hardware Objects.* 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 GC [Schoeberl 2006; Schoeberl and Vitek 2007]. 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 21 shows an example with a small object that contains two fields and an integer array of length 4. 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 instead of into the heap. This configuration is shown in the upper part of Figure 21. Note that we do not need the GC information for the hardware object handles. The Factory, which creates the hardware objects, implements this indirection.

As described in Section 5.1, we do not allow applications to create hardware objects; the constructor is private (or package visible). Figure 22 shows part of the hardware object Factory that creates the hardware object SerialPort. Two static fields (SP_PTR and SP_MTAB) are used to store the handle to the serial port 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.[8] The address of the static field SP_PTR is returned as the reference to the serial port object.

To obtain the class reference for the hardware object, we create a *normal* instance of SerialPort with new on the heap and copy the pointer to the class information. To avoid using native methods in the Factory class we delegate JVM internal work to a helper class in the JVM system package as shown in Figure 22. That helper method returns the address of the static field SP_PTR as a reference to the hardware object. All methods in class

[8]In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is XX_MTAB.

```
package com.jopdesign.io;

public class BaseFactory {

    // static fields for the handle of the hardware object
    private static int SP_PTR;
    private static int SP_MTAB;

    private SerialPort sp;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
    };

    ...

    // That's the JOP version of the JVM mechanism
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
}

package com.jopdesign.sys;

public class JVMHelp {

    public static Object makeHWObject(Object o, int address, int idx, int cp) {
        // usage of native methods is allowed here as
        // we are in the JVM system package
        int ref = Native.toInt(o);
        // fill in the handle in the two static fields
        // and return the address of the handle as a
        // Java object
        return Native.toObject(addr);
    }
}
```

Fig. 22. Part of a Factory and the helper method for the hardware object creation in the Factory

Native, a JOP system class, are *native*[9] 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 toObject(int addr) is the inverse function to map an address to a Java reference. Low-level memory access methods are used to manipulate the JVM data structures.

The main drawback of the implementation is the creation of normal instances of the hardware class. To disallow the creation with new in normal application code the visibility is set to package. However, the package visibility of the hardware object constructor is a minor issue.

To access private static fields of an arbitrary class from the system class we had to change

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

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];

static void interrupt() {

    ih[Native.rd(Const.IO_INTNR)].run();
}
```

Fig. 23. The static interrupt() method in the JVM helper class

the runtime class information: we added a pointer to the first static primitive field of that class. As addresses of static fields get resolved at class linking no such reference was needed up to now.

6.2.2 *Interrupt Handler.* Up to now JOP [Schoeberl 2005; 2008] was a very puristic hard real-time processor. There existed only one interrupt – the programmable timer interrupt as time is the primary source for hard real-time events. All I/O requests had to be handled by periodic threads that poll for pending input data or free output buffers. During the course of this paper we have added an interrupt controller to JOP and the necessary software layers.

Interrupts and exact exceptions are considered the hard part in the implementation of a processor pipeline [Hennessy and Patterson 2002]. The pipeline has to be drained and the complete processor state saved. In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [Schoeberl 2008]. On a pending interrupt (or exception generated by the hardware) we can use this translation stage to insert a special bytecode into the instruction stream. This approach keeps the interrupt completely transparent to the core pipeline.

The special bytecode (using an instruction that is unused by the JVM specification [Lindholm and Yellin 1999]) can be handled in JOP as any other bytecode: execute microcode, invoke a special method from a helper class, or execute Java bytecode from JVM.java. In our implementation we invoke the special method interrupt() from a JVM helper class.

The implemented interrupt controller (IC) is priority based. The numbers of interrupt sources can be configured. Each interrupt can also be triggered in software by a IC register write. There is one global interrupt enable and each interrupt line can be enabled and disabled locally.

The interrupt is forwarded to the bytecode/microcode translation stage including the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time.

All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable as illustrated in Figure 23.

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a class that implements Runnable and registers that class for an interrupt number. We reuse here the I/O Factory presented in Section 5.1. Figure 24 shows

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        InterruptHandler ih = new InterruptHandler();
        IOFactory fact = IOFactory.getFactory();
        // register the handler
        fact.registerInterruptHandler(1, ih);
        // enable interrupt 1
        fact.enableInterrupt(1);
        .....
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Fig. 24. An example Java interrupt handler as Runnable

a simple example of an interrupt handler implemented in Java.

For interrupts that should be handled by a sporadic thread under the control of the scheduler, following steps need to be performed on JOP:

(1) Create a SwEvent with the correct priority that performs the second level interrupt handler work
(2) Create a short first level interrupt handler as Runnable that invokes fire() of the corresponding software event handler
(3) Register the first level interrupt handler as shown in Figure 24 and start the real-time scheduler

In Section 7 we evaluate the different latencies of first and second level interrupt handlers on JOP.

6.3 Kaffe

Kaffe is an open-source[10] implementation of the JVM making it possible to add support for hardware objects and interrupt handling. Kaffe requires a fully fledged OS such as Linux to compile and run. Even though ports of Kaffe exist on uCLinux, we have not been able to find a bare bone version of Kaffe. Thus even though we managed to add support of hardware objects and interrupt handling to Kaffe, it still cannot be used without an OS.

6.3.1 *Hardware Objects.* Hardware objects have been implemented in the same manner as in the SimpleRTJ, described in Section 6.1.

6.3.2 *Interrupt Handler.* Since Kaffe runs under Linux we cannot directly support the ISR Handler approach as described in Section 4. Instead we used the ISR Thread approach in which a thread may perform a blocking call waiting for the next interrupt to occur. It turned out that the majority of implementation effort was spent in the signaling of the interrupt occurrence from the kernel space to the user space.

[10] http://www.kaffe.org/

We wrote a special Linux kernel module in the form of a character device. Through proper invocations of ioctl it is possible to ask the module to install a handler for an interrupt (e.g. the serial interrupt, normally on IRQ 7). Then the Kaffe VM can make a blocking call to read on the proper device. Finally the installed kernel handler will release the user space application from the blocked call to read upon every occurrence of the interrupt.

Using this strategy we have performed non trivial experiments implementing a full interrupt handler for the serial interrupt in Java. Still, we feel that the elaborate setup requiring a special purpose kernel device is far from our ultimate goal of running a JVM on the bare metal. Even so, the experiment has given us valuable input as to the impact of interrupt handlers and hardware objects at the Java language level.

6.4 OVM

OVM [Armbruster et al. 2007] is a research JVM implementation allowing many configurations, primarily targeted at implementation of a large subset of RTSJ while maintaining reasonable performance. OVM uses ahead of time compilation via C language: it translates both application's and VM's bytecode to C, including all classes that might be later loaded dynamically at run-time. The C code is then compiled by GCC.

6.4.1 *Hardware Objects.* Low-level hardware object support needs to provide two basic features: access to an existing hardware object from the application and creation of a hardware object from Factory methods.

To compile Java bytecode into a C program, the OVM's Java-to-C compiler first internally converts the bytecode into an intermediate representation (IR), which is still similar to the bytecode, but includes more instructions. Transformations at the IR level are both optimizations and operations necessary for correct execution, such as insertion of null-pointer checks. The produced IR is then translated to C, allowing the C compiler to perform additional optimizations. Transformations at IR level, which is similar to the bytecode, are typical also in other JVM implementations, such as Sun's HotSpot.

We base our support to access hardware objects on IR instruction transformations. We introduce two new instructions, outb and inb for byte-wide access to I/O ports. Then, we employ OVM's instruction rewriting framework to translate accesses to fields of hardware objects, putfield and getfield instructions, into sequences centered around outb and inb where appropriate. We did not implement word-wide or double-word wide access modes supported by x86 CPU. We discuss how this could be done at the end of this section.

To minimize changes needed to the OVM code, we keep the memory layout of hardware objects as if they were ordinary objects, and store port addresses into the fields representing respective hardware I/O ports. Explained with the example from Figure 19, the instruction rewriting algorithm proceeds as follows: SerialPort is a subclass of HardwareObject, hence it is a hardware object, and thus accesses to all its public volatile int fields, status and data, will be translated to port accesses to I/O addresses stored in these fields.

The translation is very simple. In case of reads, we only need to append our new inb instruction after the corresponding getfield instruction in the IR: getfield will store the I/O address on the stack and inb will replace it by a value read from this I/O address. In case of writes, we replace the corresponding putfield instruction by a sequence of swap, getfield, and outb: swap will rotate two top elements on stack, leaving the hardware object's reference on top of the stack and the value to store to I/O port below it, getfield will replace the object reference by the corresponding I/O address, and outb will send the value to the

I/O port of the given address.

The critical part of hardware object creation is to set I/O addresses into hardware object's fields. The approach is to allow a method turning off the special handling of hardware objects. In such a method, supposedly a method of the Factory that creates hardware objects, accesses to all fields of hardware objects are handled as if they were fields of regular objects. Thus, in these methods, we simply store I/O addresses to the fields.

A method can turn off the special handling of hardware objects using a marker exception mechanism, which is a natural solution within OVM. The method declares to throw PragmaNoHWIORegistersAccess exception. This exception is neither thrown nor caught, but the OVM IR level rewriter detects the declaration and disables rewriting accordingly. As the exception extends RuntimeException, it does not need to be declared in interfaces or in code calling Factory methods. In Java 1.5, not supported by OVM, a standard substitute to the marker exception would be method annotation.

Our solution depends on the representation of byte-wide registers by 16-bit fields to hold the I/O address. However, our solution could still be extended to support multiple-width accesses to I/O ports (byte, 16-bit, and 32-bit). The approach would be as follows: 32-bit I/O registers are represented by Java long fields, 16-bit I/O registers by Java int fields, and byte-wide I/O registers by Java short fields. The correct access width will be chosen by the byte-code rewriter based on the field's type.

6.4.2 *Interrupt Handler.* Low-level support heavily depends on scheduling and preemption. For our experiments, we chose the uni-processor x86 OVM configuration with green threads, running as a single Linux process. The green threads, delayed I/O operations and handlers of asynchronous events, such as POSIX signals, are only scheduled at well-defined points (*pollchecks*), which are by default at back-branches at bytecode level, at memory allocation, and at calls to the garbage collector. When no thread is ready to run, the OVM scheduler waits for events using the POSIX select call.

For handling hardware interrupts in OVM we use Xenomai RT Linux [Xenomai developers 2008; Gerum 2004]. Xenomai allows implementation of real-time systems in user-space Linux. Xenomai tasks, which are in fact user-space Linux threads, can run either in the Xenomai primary domain, or in the Xenomai secondary domain. In the primary domain, they are scheduled by the Xenomai scheduler, isolated from the Linux kernel. In the secondary domain, Xenomai tasks behave similarly to regular real-time Linux threads. Tasks can switch to the primary domain at any time, but are automatically switched back to the secondary domain whenever they invoke a Linux system call. A single Linux process can have threads of differing types: regular Linux threads, Xenomai primary domain tasks, and Xenomai secondary domain tasks. In the primary domain tasks can wait on hardware interrupts with a higher priority than the Linux kernel. The Xenomai API provides the interrupts using the ISR Thread approach and supports *virtualization* of basic interrupt operations – disabling and enabling the interrupt or all local CPU interrupts. These operations have the same semantics as real interrupts have and disabling/enabling a particular operation leads to the corresponding operation at hardware level.

Before our extension, OVM ran as a single Linux process with a single (native Linux) thread, a *main OVM thread*. This native thread implemented Java green threads. To support interrupts, we add additional threads to the OVM process: for each interrupt source handled in OVM, we dynamically add a interrupt listener thread, running in the Xenomai primary domain.

Upon receiving an interrupt, the listener thread marks the pending interrupt in a global memory structure shared with the main OVM thread. Once the OVM thread reaches a pollcheck, it discovers that an interrupt is pending, preempts the current Java green thread and enters OVM's scheduler. The scheduler then, again using the ISR Thread approach, immediately wakes-up and schedules the Java green thread that is waiting for the interrupt. To simulate the first level ISR handler approach, this green thread then invokes some handler method. In a non-RTSJ scenario, the green thread invokes the run() method of the associated InterruptHandler (see Figure 17). In an RTSJ scenario, a specialized thread fires an asynchronous event bound to the particular interrupt source. It invokes the fire() method of the respective RTSJ's AsyncEvent. As mentioned in Section 5.3, the RTSJ logic of AsyncEventHandler (AEH) registered to this event should be again an instance of InterruptHandler, in order to allow the interrupt handling code to access basic interrupt handling operations.

To provide an InterruptHandler instance with the same semantics as a real first-level handler, we virtualize the interrupt handling operations for interrupt enabling, disabling, etc. Therefore, we have two levels of interrupt virtualization, one is provided by Xenomai to our listener thread, and the other is, on top of the first one, provided by the OVM runtime to the InterruptHandler instance. In particular, local CPU interrupts are emulated, hardware interrupts are disabled/enabled at interrupt controller level, interrupt ending is performed at interrupt controller level (via the Xenomai API), and interrupt starting is emulated (it only acknowledges the listener thread that the interrupt was received).

The RTSJ scheduling features (deadline checking, inter-arrival time checking, delaying of sporadic events) related to release of the AEH should not require any further adaptations for interrupt handling. We however could not test these features as OVM does not implement them.

OVM uses *thin monitors*, which means that a monitor is only instantiated (*inflated*) when a thread has to block on acquiring it. This semantic however does not match what we need – disable the interrupt when the monitor is acquired to prevent the handler from interrupting. Our solution is thus that we provide a special implementation of a monitor for interrupt handlers and inflate it in the constructor of InterruptHandler. This way we do not slow down synchronization in non-interrupt handling code.

6.5 Summary

To add support for hardware objects (see Section 3) and interrupt handling (see Section 4) to all four JVMs some common techniques have been applied. Accessing device registers through hardware objects can be achieved by extending the behavior of the bytecodes putfield and getfield or redirecting the pointer to the object. These bytecodes can be extended to identify if the field being accessed is inside a hardware object. If so, the implementation of the actual access is implemented differently. Similarly, the implementation of interrupt handling requires changes to the bytecodes monitorenter and monitorexit; they must be extended to identify if the object being used for locking is an interrupt handler object. If so, the implementation of the actual locking must be changed to disable/enable interrupts. Whether dealing with hardware or interrupt objects, we used the same approach of letting the hardware object and interrupt handler classes inherit from the super classes HardwareObject and InterruptHandler respectively.

For JVMs that need a special treatment of bytecodes putfield and getfield (SimpleRTJ, Kaffe, and OVM) bytecode rewriting at runtime can be used to avoid the additional check

of the object type. This is a standard approach (called *quick* bytecodes in the first JVM specification) in JVMs to speedup field access of resolved classes.

Historically, registers of most x86 I/O devices are mapped to a dedicated I/O address space, which is accessed using dedicated instructions - port read and port writes. Fortunately, both the processor and Linux allow user-space applications running with administrator privileges to use these instructions and access the ports directly, via iopl, inb, and outb calls. For both the Kaffe and OVM implementations we have implemented bytecode instructions putfield and getfield accessing fields of a hardware object by calls to iopl, inb, and outb.

Linux does not allow user-space applications to handle hardware interrupts. Only kernel space functionality is allowed to register interrupt handlers. We have overcome this issue in two different ways:

—For OVM we have used the Xenomai real-time extension to Linux. Xenomai extends the Linux kernel to allow for the creation of real-time threads and allows user space code to wait for interrupt occurrences.

—For Kaffe we have written a special purpose kernel module through which the user space application (the Kaffe VM) can register interest in interrupts and get notified about interrupt occurrence.

Both these workarounds are intended to allow for an incremental transition of the JVMs and the related development libraries into a direct (bare bone) execution environment. In that case the workarounds would no longer be needed.

If a compiling JVM is used (either as JIT or ahead-of-time) the compiler needs to be aware of the special treatment of hardware objects and monitors on interrupt handlers. One issue, which we did not face in our implementations, could be the alignment of object fields. When device registers are represented by different sized integer fields the compiler needs to pack the data structure.

The restrictions within an interrupt handler are JVM dependent. If an interruptible, real-time GC is used (as in OVM and JOP) objects can be allocated in the handler and the object graph may be changed. For a JVM with a stop-the-world GC (SimpleRTJ and Kaffe) this operations are not allowed as the handler can interrupt the GC.

7. EVALUATION AND CONCLUSION

Besides implementing the Java HAL in four different JVMs we evaluate the HAL with a demonstrator and measure the performance of hardware access via hardware objects and the latency of Java interrupt handlers.

7.1 Demonstrator

For first tests we implemented a serial port driver with hardware objects and interrupt handlers. As the structure of the I/O registers are exactly the same on a PC, the platform for SimpleRTJ, and JOP we were able to use the exact same definition of the hardware object SerialPort and the test programs on all four systems.

Using again the serial device we run an embedded TCP/IP stack, implemented completely in Java, over a SLIP connection. The TCP/IP stack contains a tiny web server and we were able to serve web pages with a Java only solution as shown in the introduction in Figure 6. It has to be noted that the TCP/IP stack, the tiny web server, and the hard-

	JOP		OVM		SimpleRTJ		Kaffe	
	read	write	read	write	read	write	read	write
native	5	6	5517	5393	2588	1123	11841	11511
HW Object	13	15	5506	5335	3956	3418	9571	9394

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

ware object for the serial port are the same for all platforms. The only difference is in the hardware object creation with the platform dependent Factory implementations.

7.2 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 goal for relatively slow embedded processors. It matters less on general purpose processors, where the slow I/O bus essentially limits the access time.

7.2.1 *Measurement Methodology.* Execution time measurement of single instructions is only possible on simple in-order pipelines when a cycle counter is available. Even on simple pipelines execution time dependencies (e.g. load-use dependency) exist. On a modern super-scalar architecture where hundreds of instructions are in flight each clock cycle direct execution time measurement becomes impossible. Therefore, we performed a bandwidth based measurement. We measure how many I/O instructions per second can be executed in a tight loop. The benchmark program is self-adapting and increases the loop count exponentially till the measurement run for more than one second and the iterations per second are reported. To compensate for the loop overhead we also perform an overhead measurement of the loop and subtract that overhead from the I/O measurement.

Figure 25 shows the measurement loop for the I/O read operation in method test() and the overhead loop in method overhead(). In the comment above the method the bytecodes of the loop kernel is shown. We can see that the difference between the two loops is the single bytecode getfield that performs the I/O read request.

The iteration count cnt is increased by the benchmark framework until method test() run for more than a second. From this execution time the time measured for method overhead() with the same iteration count is subtracted and the I/O bandwidth b obtained in operations per second

$$b = \frac{cnt}{t_{test} - t_{ovhd}}$$

7.2.2 *Execution Time.* In Table IV we compare the access time to a device register with native functions to the access via hardware objects. The execution time is given in clock cycles. We scale the measured I/O bandwidth b with the clock frequency f of the system under test by $n = \frac{f}{b}$.

We run the measurements on a 100 MHz version of JOP. As JOP is a simple pipeline we can also measure short bytecode instruction sequences with the cycle counter. Those measurements provided the exact same values as the one given by our benchmark validating our benchmark approach. 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 the indirection through the handle for the field access.

```
public class HwoRead extends BenchMark {

    SysDevice sys = IOFactory.getFactory().getSysDevice();

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ALOAD 2
    GETFIELD com/jopdesign/io/SysDevice.uscntTimer : I
    IADD
    ISTORE 3
*/
    public int test(int cnt) {

        SysDevice s = sys;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+s.uscntTimer;
        }
        return a;
    }

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ILOAD 2
    IADD
    ISTORE 3
*/
    public int overhead(int cnt) {

        int xxx = 456;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+xxx;
        }
        return a;
    }
}
```

Fig. 25. Benchmark for the I/O read operation measurement

Despite the slower I/O access via hardware objects on JOP the access is fast enough for all currently available I/O devices. Therefore, we currently change all device drivers to use hardware objects.

The measurement for OVM was run on a Dell Precision 380 (Intel Pentium 4, 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux 2.6 (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). OVM was compiled without Xenomai support and the generated virtual machine was compiled with all optimizations enabled. As I/O port we used the printer port. Access to the I/O port via a hardware object is just slightly faster than access via native methods. This was expected as the slow I/O bus dominates the access time.

On the SimpleRTJ JVM the native access is faster than access to hardware objects. The reason is that the SimpleRTJ JVM does not implement JNI, but has its 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, using hardware objects need a field lookup that is more time consuming than invoking a static method. With bytecode-level optimization at class load time it would be possible to avoid the expensive field lookup.

We measured the I/O performance within Kaffe on an Intel Core 2 Duo T7300, 2.00 GHz with Linux 2.6.24 (Fedora Core 8). We used access to the serial port for the measurement. On the interpreting Kaffe JVM we notice a difference between the native access and hardware object access. Hardware objects are around 20% faster.

7.2.3 *Summary.* For practical purposes, the overhead on using hardware objects is insignificant. In some cases there may even be an improvement in performance. The benefits in terms of safe and structured code should make this a very attractive option for Java developers.

7.3 Interrupt Handler Latency

7.3.1 *Latency on JOP.* To measure interrupt latency on JOP we use a periodic thread and a interrupt handler. The periodic thread records the value of the cycle counter and triggers the interrupt. In the handler the counter is read again and the difference between the two is the measured interrupt latency. A plain interrupt handler as Runnable takes a constant 234 clock cycles (or 2.3 μs for a 100 MHz JOP system) between the interrupt occurrence and the execution of the first bytecode in the handler. This quite large time is the result of two method invocations for the interrupt handling: (1) invocation of the system method interrupt() and (2) invocation of the actual handler. For more time critical interrupts the handler code can be integrated in the system method. In that case the latency drops down to 0.78 μs. For very low latency interrupts, the interrupt controller can be changed to emit different bytecodes depending on the interrupt number. In that case we can avoid the dispatch in software and can implement the interrupt handler in microcode.

We have integrated the two-level interrupt handling at the application level. We setup two threads: one periodic thread, that triggers the interrupt and a higher priority event thread that acts as second level interrupt handler and performs the handler work. The first level handler just invokes fire() for this second level handler and returns. The second level handler gets scheduled according to the priority. With this setup the interrupt handling latency is 33 μs. We verified this time by measuring the time between fire of the software event and the execution of the first instruction in the handler directly from the periodic

	Median (us)	3rd Quartile (us)	95% Quantile (us)	Maximum (us)
Polling	3	3	3	8
Hard	14	16	16	21
Kernel	14	16	16	21
User	17	19	19	24
Ovm	59	59	61	203

Fig. 26. Interrupt (and polling) latencies in microseconds.

thread. This time took 29 μs and is the overhead due to the scheduler. This measurement is consistent with the measurements in [Schoeberl and Vitek 2007]. There we measured a minimum useful period of 50 μs for a high priority periodic task.

The runtime environment of JOP contains a concurrent real-time GC [Schoeberl and Vitek 2007]. The GC can be interrupted at a very fine granularity. During sections that are not preemptive (data structure manipulation for a new and write-barriers on a reference field write) interrupts are simply turned off. The copy of objects and arrays during the compaction phase can be interrupted by a thread or interrupt handler [Schoeberl and Puffitsch 2008]. Therefore, the maximum blocking time is in the atomic section of the thread scheduler and not in the GC.

7.3.2 *Latency on OVM/Xenomai.* For measuring OVM/Xenomai interrupt latencies, we have extended an existing interrupt latency benchmark written by Jan Kiszka from the Xenomai team [Xenomai developers 2008]. The benchmark uses two machines connected over serial line. The *log* machine, running a regular Linux kernel, toggles the RTS state of the serial line and measures the time it takes for the *target* machine to toggle it back.

To minimize measuring overhead, the *log* machine uses only polling and disables local CPU interrupts while measuring. Individual measurements are stored into memory and dumped at shutdown, so that they can be analyzed offline. We have made 400,000 measurements in each experiment, reporting only the last 100,000 (this was to warm-up the benchmark, including memory storage for the results). The *log* machine toggles the RTS state regularly with a given period. We have run three sets of experiments, for 100, 150, and 200 us periods.

We have tested 5 versions of the benchmark on *target* machine: polling version written in C (*polling*), interrupt handler written in Java/OVM/Xenomai (*ovm*), user-space interrupt handler written in C/Xenomai (*user*), kernel-space interrupt handler in C/Xenomai running out of control of the Linux scheduler (*kernel*), and hard-realtime kernel-space interrupt handler running out of control of both the Xenomai scheduler and the Linux scheduler (*hard*).

Results for 150 us period are shown in Figure 26. The median overhead for interrupt handling over polling is about 11 us. The additional overhead for handling interrupts in user space is 3 us. The additional overhead of doing this in the present version of OVM is 42 us. Median overhead for using the Xenomai scheduler is not measurable.

The maximum latency of OVM with the 150 us period was 101 us, due to infrequent pauses. Their frequency is so low that the measured 99% quantile is only 69 us. The other *log* machine periods, 100 us and 200 us, reported similar results.

The experiment was run on Dell Precision 380 (Intel Pentium 4 3.8GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux 2.6 (Ubuntu 7.10, Linux 2.6.24.3 with

Xenomai-RT patch). As Xenomai is still under active development, we had to use Xenomai workarounds and bugfixes, mostly provided by Xenomai developers, to make OVM/Xenomai work.

7.3.3 *Summary.* The overhead for implementing interrupt handlers is very acceptable, since interrupts are used to signal relatively infrequently occurring events like end of transmission, loss of carrier etc. With a reasonable work division between first level and second level handlers, the proposal does not introduce dramatic blocking terms in a real-time schedulability analysis, and thus it is suitable for embedded systems.

7.4 Discussion

7.4.1 *Safety 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.

7.4.2 *Portability.* 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.

7.4.3 *Compatibility with the RTSJ Standard.* As shown for the OVM implementation, the proposed HAL is compatible with the RTSJ standard. We consider it to be a very important point, since many existing systems have been developed using such platforms or subsets thereof. In further development of such applications existing and future interfacing to devices may be refactored using the proposed HAL. It will make the code safer and more structured and may assist in possible ports to new platforms.

7.5 Perspective

This demonstrator shows that we achieved an almost platform independent representation of the hardware and it is possible to implement system level functionality in Java. As future work we will add devices drivers for common I/O devices such as network interfaces[11] and hard disc controllers. On top of these drivers we will implement a file system and other typical OS related services towards our final goal of a Java only system.

An interesting question is if we can define a common set of *standard* hardware objects. The SerialPort was a lucky example – although the internals of the JVMs and the hardware was different we had a compatible hardware object that worked on all platforms. It should be possible that a chip manufacturer provides, beside the data sheet that describes the registers, a Java class for the register definitions of that chip. This definition can be reused in all systems that use that chip.

Another interesting idea is to define the interaction between the GC and hardware objects. We stated that the GC should not collect hardware objects. If we relax this restriction we can redefine the semantics of collecting an object: on running the finalizer for a hardware object the device can be put into sleep mode.

[11] A device driver for a CS8900 based network chip is already part of the Java TCP/IP stack.

ACKNOWLEDGMENTS

We wish to thank Andy Wellings for his insightful comments on an earlier version of the paper. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

REFERENCES

aJILE. 2000. aj-100 real-time low power Java processor. preliminary data sheet.

ARMBRUSTER, A., BAKER, J., CUNEI, A., FLACK, C., HOLMES, D., PIZLO, F., PLA, E., PROCHAZKA, M., AND VITEK, J. 2007. A real-time java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys. 7,* 1, 1–49.

BACON, D. F., CHENG, P., AND RAJAN, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 285–298.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. Java Series. Addison-Wesley.

BURNS, A. AND WELLINGS, A. J. 2001. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc.

CASKA, J. micro [μ] virtual-machine. http://muvium.com/.

CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. 2001. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev. 35,* 5, 73–88.

FELSER, M., GOLM, M., WAWERSICH, C., AND KLEINÖDER, J. 2002. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*. 45–58.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional.

GERUM, P. 2004. Xenomai - implementing a RTOS emulation framework on GNU/Linux. `http://http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf`.

GROUP, T. C. 2008. Trusted computing. available at https://www.trustedcomputinggroup.org/.

HANSEN, P. B. 1977. *The Architecture of Concurrent Programs*. Prentice-Hall Series in Automatic Computing. Prentice-Hall.

HENNESSY, J. AND PATTERSON, D. 2002. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303.

HUNT, G., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. D. 2005. An overview of the singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research (MSR). Oct.

KORSHOLM, S., SCHOEBERL, M., AND RAVN, A. P. 2008. Java interrupt handling. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. IEEE Computer Society, Orlando, Florida, USA.

KRALL, A. AND GRAFL, R. 1997. CACAO – A 64 bit JavaVM just-in-time compiler. In *PPoPP'97 Workshop on Java for Science and Engineering Computation*, G. C. Fox and W. Li, Eds. ACM, Las Vegas.

KREUZINGER, J., BRINKSCHULTE, U., PFEFFER, M., UHRIG, S., AND UNGERER, T. 2003. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems 27,* 1, 19–31.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison-Wesley, Reading, MA, USA.

LOHMEIER, S. 2005. Jini on the jnode java os. Online article at http://monochromata.de/jnodejini.html.

RAVN, A. P. 1980. Device monitors. *IEEE Transactions on Software Engineering 6,* 1 (Jan.), 49–53.

RTJ COMPUTING. 2000. simpleRTJ a small footprint Java VM for embedded and consumer devices. online at http://www.rtjcom.com/.

SCHOEBERL, M. 2005. Jop: A java optimized processor for embedded real-time systems. Ph.D. thesis, Vienna University of Technology.

SCHOEBERL, M. 2006. 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)*. Gyeongju, Korea, 424–432.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2*, 265–286.

SCHOEBERL, M., KORSHOLM, S., THALINGER, C., AND RAVN, A. P. 2008. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. IEEE Computer Society, Orlando, Florida, USA.

SCHOEBERL, M. AND PUFFITSCH, W. 2008. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*.

SCHOEBERL, M. AND VITEK, J. 2007. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. ACM Press, Vienna, Austria, 85–93.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39,* 9, 1175–1185.

SIEBERT, F. 2002. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books.

SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. 2006. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*. ACM Press, New York, NY, USA, 78–88.

WILKINSON, T. 1996. Kaffe – a virtual machine to run java code. Available at http://www.kaffe.org.

WIRTH, N. 1977. Design and implementation of modula. *Software - Practice and Experience 7*, 3–84.

WIRTH, N. 1982. *Programming in Modula-2*. Springer Verlag.

XENOMAI DEVELOPERS. 2008. Xenomai: Real-time framework for Linux. `http://www.xenomai.org`.

Received August 2008; revised Month Year; accepted Month Year