Interrupt Handlers in Java

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

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

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

Abstract

An important part of implementing device drivers is to control the interrupt facilities of the hardware platform and to program interrupt handlers. Current methods for handling interrupts in Java use a server thread waiting for the VM to signal an interrupt occurrence. It means that the interrupt is handled at a later time, which has some disadvantages. We present constructs that allow interrupts to be handled directly and not at a later point decided by a scheduler. A desirable feature of our approach is that we do not require a native middelware layer but can handle interrupts entirely with Java code. We have implemented our approach using an interpreter and a Java processor, and give an example demonstrating its use.

1 Introduction

In an embedded system which is implemented as a collection of cooperating threads, interrupt handlers are more similar to subroutines than to threads. The handlers are not scheduled as threads; they may be invoked any time and even interrupt the thread scheduler, because interrupts are dispatched by the hardware as response to external events.

Current profiles for real-time Java [5, 2] attempt to hand over interrupts to normal sporadic threads as quickly as possible. This can be done by installing a piece of native code, called the first level interrupt handler, to be invoked by the hardware, and let this code register the interrupt occurrence and then return. Then at each reschedule point the interrupt mark is checked and a waiting thread is unblocked to handle the interrupt (this is sometimes called the second level interrupt handler). Examples of this approach are found in e.g. the Squawk VM [11] and the JamaicaVM from aicas [1]. Squawk reports an average latency of 0.1 milliseconds and a worst case latency of around 13 milliseconds. From [4] we see that the JamaicaVM has an average latency of 50 microseconds and a worst case latency of 250 microseconds. In both cases the interrupts are handled out-of-context.

An advantage of this approach is that the execution of the second level interrupt handler is controlled by the normal thread scheduler and thus observes the priorities and scheduling principles of it. Less desirable features are:

Latency: A delay is introduced from the occurrence of the interrupt until the point in time when it is handled.

Out-of-context execution: The interrupt is handled out of the context of the first level interrupt handler. This means that any computation that requires this context cannot be done in Java, but must be done in native code. If the context dependent part can be kept stereotypical, this is less of a problem. In other cases where device dependent actions are needed, the native middelware layer becomes complex.

The contribution of this paper consists in the design and implementation of a mechanism for handling interrupts in Java. It does not hand over the interrupt to a sporadic thread, but handles the interrupt completely in the context of the first level interrupt handler. We call this mechanism *in-context interrupt handling*. An important feature of in-context interrupt handling is that actions that need to be done in the context of the first level interrupt handler can now be done in Java, in effect eliminating the need for a native middelware layer. This mechanism does not preclude the standard way of handling interrupts in real-time Java, but complements it and can be used in cases where the standard method is inadequate for one of the reasons given above.

In Section 2 we will describe how interrupts are handled in legacy systems. In Section 3 we introduce our design for in-context interrupt handling, and discuss how the method can be supported in existing Java execution environments. In Section 4 we demonstrate how the design has been implemented for two different execution environments for Java: the JOP Java processor and the SimpleRTJ interpreter. Then in Section 5 we show a simple example of using our interrupt handling implementations. We conclude the paper in Section 6.

```
volatile uint16 P0_UART_RX_TX_REG @ 0xFFE032;
volatile uint16 P0_CLEAR_RX_INT_REG @ 0xFFE036;
volatile uint16 RESET_INT_PENDING_REG @ 0xFFE202;

#define CLR_UART_RX_INT_PENDING 0x0010
#define CLEAR_UART_RI_FLAG P0_CLEAR_RX_INT_REG = 0
#define CLEAR_PENDING_UART_RI_INTERRUPT \
    RESET_INT_PENDING_REG = CLR_UART_RX_INT_PENDING

__interrupt void Uart_RX_Interrupt(void) {
   UartRxBuffer[UartRxWrPtr++] = P0_UART_RX_TX_REG;
   if (UartRxWrPtr>=sizeof(UartRxBuffer)) {
      UartRxWrPtr=0;
   }
   CLEAR_UART_RI_FLAG;
   CLEAR_PENDING_UART_RI_INTERRUPT;
}
```

Figure 1. An example interrupt handler in C

2 Conventional Interrupt Handling

Interrupts are used to signal external events for example, detecting that a button has been pressed. When an interrupt occurs the processor simply stops executing the code it runs, and jumps to an interrupt routine instead. The jump saves the environment of the interrupted process so that it may be restored later; this includes saving the CPU registers and the processor status register. This makes it possible to continue the execution of the original code when the interrupt routine has been executed.

Saving the interrupted context and setting up a new context for the interrupt handler is called a *context switch*. Some hardware platforms implement a *full* context switch in hardware, where other platforms implements a *partial* context switch in hardware. In the latter case the programmer must save those parts of the interrupted context that he needs to overwrite in the interrupt handler.

As an example interrupt handler, Figure 1 shows an excerpt of code implementing the RS232 receive interrupt for an existing legacy system. The RS232 receive interrupt is generated by the hardware when the RS232 interface receives a byte on the serial line. It is the job of the interrupt handler to retrieve the byte from the proper device register and clear the receive interrupt flag.

Though this example contains non-standard compiler directives and runs on a particular piece of hardware, it illustrates the following general features:

I/O Memory: Through the compiler directive name @ address the name, e.g. P0_UART_RX_TX_REG is designated to refer directly to a physical memory location, in this case one of the memory mapped device registers of the UART. Any assignment or query of these names in the code will correspond to reads and writes of the particular register.

Interrupt handlers: Through the compiler directive __interrupt the function void Uart_RX_Interrupt(void) becomes an interrupt routine. This basically means that exit and entry code is generated by the compiler to save and restore the state of the interrupted process.

The circular buffer UartRxBuffer can be read by user code outside the context of the interrupt to handle the bytes received. Some kind of mutual exclusion between user code and the interrupt handler may be required. This is typically implemented by disabling interrupts.

3 In-context Interrupt Handling

Our goal is to be able to implement the interrupt handler from Figure 1 in pure Java. The two important tasks that the Uart_RX_Interrupt handler must do are:

1. Retrieve the received byte from the proper device register and save the byte in a data structure.
2. Clean up from the interrupt, in this case clear the UART receive interrupt flag (CLEAR_UART_RI_FLAG; CLEAR_PENDING_UART_RI_INTERRUPT).

Using the RTSJ profile [2] for Java the above tasks can naturally be solved in the following manner:

Ad 1) Using the raw memory access supplied by RTSJ through the class RawMemoryAccess, it is possible to read the received byte from the proper device register. A detailed example on how this looks is available in [3] (Chapter 15.5).

Ad 2) The suggested way to handle interrupts in RTSJ is to use the AsyncEvent and AsyncEventHandler classes. Again [3] includes a detailed example. In the RTSJ an interrupt occurring is equivalent to the fire method being called on the AsyncEvent object. This in turn will call the run() method on all installed handlers. But, to handle the interrupt from Figure 1 in pure Java, the fire method *must be called in the context of the interrupt*. The reason is that the receive interrupt flag must be cleared before exiting the interrupt context. Failing to do so will cause the interrupt to recur. In all implementations of RTSJ that we know of, handling of the AsyncEvent corresponding to the interrupt will be scheduled outside the context of the interrupt. This does not allow us to implement the handler from Figure 1 in pure Java.

As a complementary way to handle interrupts in Java we suggest that *the native code implementing the first level interrupt handler is used to call the JVM and start executing the appropriate interrupt handler immediately, or in other words, before returning from the interrupt*. It makes it possible to handle the interrupt completely in its context. Thus, we will be able to implement the example in Figure 1 in pure Java, which includes to clear the interrupt receive flag from inside Java code.

Whether it is possible to reenter the JVM inside the context of the first level interrupt handler in order to execute the Java part of the interrupt handler depends on the scheduling mechanism and the GC strategy. In the remainder of this section we will look at the consequences of the proposal for different types of Java execution environments.

3.1 Scheduling

If native threads are used and attached to the VM e.g through the JNI [6] function JNI_AttachCurrentThread it should be straightforward to reenter the JVM while it is interrupted, because from the point of view of the JVM the interrupt handler is not different from a normal high priority thread that has been switched in by the external scheduler.

If an internal scheduler is used (also called *green threads*) it will most likely require some work to refactor the JVM implementation to support reentry at any time. The reason is that the JVM implementation knows when thread switching can occur and explicitly or implicitly has used this knowledge when accessing global data. The SimpleRTJ VM [7], used for one of the experiments described in Section 4, includes an internal scheduler and the section shows the work required to make the JVM reenterable.

3.2 Garbage Collection

When executing the Java first level interrupt handler[1] in the context of the interrupt, it becomes very important that the handler is short lived. The reason for this restriction is that while an interrupt handler is executing, no other interrupts of the same type can be handled. In many cases no other interrupts at all can be handled, thus making it particularly important to complete the interrupt handler swiftly. In particular, this means that the interrupt handler cannot block waiting for a thread.

In normal execution environments for Java, threads synchronize around garbage collection (GC) to avoid disturbing an ongoing GC. In the case of interrupt handlers, this become impossible. Fruthermore, it is not feasible to let interrupt handlers start a lengthy GC process. Both these facts affect the interoperability of interrupt handlers with a GC.

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 do an accurate collection will not change during the collection. Using stop-the-world collection an interrupt handler may 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.

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. Using concurrent collection it should be possible to allow for allocation of objects 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 should be known.

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 thread switching and 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.

4 Supporting Interrupt Handlers

To experiment with our design for in-context interrupt handling we have added such support to the SimpleRTJ interpreter [7] and the experimental Java processor JOP.

4.1 Interrupt Handlers in SimpleRTJ

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. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 3.2.

Reentering the JVM

The SimpleRTJ JVM uses green threads. This means that it had to be refactored quite a bit to allow for reentering the JVM from inside the first level interrupt handler. What we did was to get 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 shared data we pass around a single pointer to the shared data now allocated on the stack.

Context Switching at Interrupt

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 waiting for events to occur. This server thread runs at highest priority. The SimpleRTJ JVM reschedule points are in between the execution of each bytecode. This means that 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

[1] In this section when we use the term "interrupt handler" we mean an interrupt handler executed in-context as described in Section 3

event. To achieve in-context interrupt handling 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 to get rid of the global state. How difficult this is, 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 RS232 receive interrupt. The final receive interrupt handler implemented in pure Java is shown in Section 5.

4.2 Interrupt Handlers on JOP

We have implemented a priority based interrupt controller in JOP. The numbers of interrupt lines can be configured. An interrupt can also be triggered in software. There is one global interrupt enable and a local enable for each interrupt line.

In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [8]. 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 trick keeps the interrupt completely transparent to the core pipeline. Interrupts are accepted at bytecode boundaries and clear the global enable flag when accepted. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. Therefore, the execution of the interrupt handler starts with global disable. The interrupts have to be enabled again by the handler at a *convenient* time.

The special bytecode 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.

Interrupt Handling

All interrupts are mapped to one 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. The timer interrupt (index 0) is handled specially. On a timer interrupt the real-time scheduler of JOP gets invoked. At system startup the table of Runnables is initialized with a no-op handler.

Applications provide handlers via objects that implements Runnable and register the object for a interrupt number. We reuse here the I/O Factory presented in [9]. Figure 2 shows a simple example of an interrupt handler implemented in Java.

```
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!");
    }
}
```

Figure 2. An interrupt handler as Runnable

For interrupts that should be handled by a sporadic thread under the control of the scheduler, the following needs to be performed on JOP: (1) Create a SwEvent (similar to the RTSJ AsyncEventHandler) that performs the second level interrupt handler work; (2) create a short first level interrupt handler as Runnable and invoke fire() of the corresponding software event handler; (3) register the first level interrupt handler as shown in Figure 2 and start the real-time scheduler.

Garbage Collection

The runtime environment of JOP contains a concurrent real-time GC [10]. The GC can be interrupted at a very fine grain level. During sections that are not preemptive (e.g. data structure manipulation for a new, write barriers on reference field write, object copy during compaction) interrupts are simply turned off. The longest blocking time due to the GC work is on an object or array copy. In [10] we have observed maximum blocking times of 40 μs induced by the GC with medium sized arrays.

5 Using Interrupt Handler

We have not seen any need for adding to the RTSJ style of programming with interrupts (described in Section 3). We have just changed the way that the AsyncEvent gets scheduled. In our approach the server thread bound to the handling of the event gets released immediately inside the context of the first level interrupt handler and not at some later point. Using the skimmed down version of the javax.events package distributed with the SimpleRTJ JVM, the legacy interrupt handler for the RS232 receive interrupt illustrated in Figure 1, can be translated into pure Java as it is shown in Figure 3.

5.1 Accessing Device Registers

A very important part of what interrupt handlers normally need to do is to access device registers. To perform this access efficiently, which is a requirement for interrupt handlers,

```
public class RS232ReceiveInterruptHandler
  extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;
    private short UartRxBuffer[];
    private byte UartRxWrPtr;

    public RS232ReceiveInterruptHandler(RS232 rs232,
                InterruptControl interruptControl) {
      super(INT_RS232RX); // Subscribe to the UART receive int.
      this.rs232 = rs232;
      this.interruptControl = interruptControl;
      UartRxBuffer = new short[32];
      UartRxWrPtr = 0;
    }
    protected void handleInterrupt() {
      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;
    }
}
```

Figure 3. An example RS232 interrupt handler

we use hardware objects as defined in [9]. The hardware objects rs232 and interruptControl has been defined to fit the physical hardware platform and allows the interrupt handler to access appropriate device registers directly.

6 Conclusion

We have introduced the concept of in-context interrupt handling and shown its implementation in an interpreter and on a Java processor. An example shows that in-context interrupt handling allows for a greater portion of the interrupt handler to be written in Java. On legacy systems implemented in C/assembler the default is for interrupt handlers to be executed in-context, so adding this option as well on Java based systems will seem natural to experienced programmers of embedded systems.

The proposal has an impact on the safety, portability and maintainability of an application. It is clear that Java code with interrupt handlers may bring the system down, but that is not different from having the handlers in middleware. Yet, the basic safety features of Java (pointer checks, index checks, type checking) are with the proposal brought to bear on such low level code and thus the safety is improved. Interrupt handlers are highly platform dependent and not portable; but they are essential for applications, so placing them outside the Java application only seemingly makes it portable. With the good structuring facilities of packages, classes and interfaces, a well-architected application will preserve portability by placing interrupt handlers in separate hardware abstraction packages. Finally, maintainability will be improved by having one language for an application, where common documentation standards are more likely to be applied.

The current proposal comes in this paper with a proof of concept, but in order for it to really succeed, it needs at some point in time to enter as part of a standard profile and most importantly be included in the JVM platforms.

Acknowledgements

We are grateful to the reviewers for their useful comments.

References

[1] aicas. http://www.aicas.com/jamaica.html. Visited June 2007.

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

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

[4] J. M. Enery, D. Hickey, and M. Boubekeur. Empirical evaluation of two main-stream rtsj implementations. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 47–54, New York, NY, USA, 2007. ACM.

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

[6] S. Liang. *The Java Native Interface - Programmers Guide and Specification*. Addison-Wesley, 1999.

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

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

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

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

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