

## Experience in Building a Real-Time Extension Library for Java\*

HSIN-TA CHIAO, SCOTT HSU-JING KAO<sup>+</sup>, YUE-SHAN CHANG<sup>++</sup>,  
SHEN-TZAY HUANG<sup>+</sup> AND SHYAN-MING YUAN

*Department of Computer and Information Science  
National Chiao Tung University  
Hsinchu, 300 Taiwan*

*E-mail: {gis84532, smyuan}@cis.nctu.edu.tw*

<sup>+</sup>*Department of Computer and Information Science  
National Pingtung University of Science and Technology  
Pingtung, 912 Taiwan*

*E-mail: {m8856004, sthuang}@mail.npust.edu.tw*

<sup>++</sup>*Department of Electronic Engineering*

*Ming-Hsin Institute of Technology*

*Hsinchu, 304 Taiwan*

*E-mail: ysc@mhit.edu.tw*

For building real-time control programs on PC controllers, we designed and implemented a real-time extension library to enhance the Java virtual machine that was already available in the real-time operating system we used. Our extension library has the following advantages: First, the underlying Java virtual machine needs no modification to accommodate it. Second, this extension library is easily ported to any other priority-based real-time operating system. Third, the core of this extension library is basically derived from a subset of the *Real-Time Specification for Java* (RTSJ) standard, and thus our real-time control program can be moved to a forthcoming RTSJ-compliant Java virtual machine without much difficulty. In brief, our work may help suggest ways of implementing RTSJ, or the notion of (degrees of) “minimal” compliance/support of RTSJ based on non-RTSJ Java virtual machines. In this paper, we will show the requirement, application programming interface, and implementation of this extension library, and discuss its influence on timing.

**Keywords:** Java<sup>TM</sup>, RTSJ, real-time systems, asynchronous event handling, POSIX

### 1. INTRODUCTION

Java technologies, including the Java programming language and the Java Virtual Machine (JVM), have achieved broad acceptance in the developer community since their introduction in 1995. Java’s simplified object model, strong notions of safety and security, integral multithreading support, and promise of Write Once, Run Anywhere (WORA) have much to offer real-time and embedded developers. However, the large size, non-deterministic behavior and poor performance of most Java implementations have

---

Received September 3, 2001; accepted April 15, 2002.

Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.

\* A preliminary version of this paper was presented in the 2001 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2001), Toronto, Canada, May 13-16, 2001. Besides, this research was supported by the National Science Council grant 89-2218-E-009-032, the industry research program 89-EC-2-A-17-0285-006 of the ROC Economic Bureau, and the Ministry of Education’s Program of Excellence Research 89-EFA04-1-4.

hampered itself in the real-time and embedded communities.

Real-time programming is a critical component in the development of many consumer, industrial, and system devices. In using Java to build a real-time control system, we found the following problems: First, each Java thread can be assigned a scheduling priority; however, the specification for Java does not strictly define the scheduling semantics among Java threads [1, 2]. That is naturally viewed as an implementation-dependent issue, so a real-time Java program could not have the characteristic of WORA like general Java programs. Second, the Java garbage collector cannot be scheduled, and it may randomly block the execution of all threads inside a Java virtual machine [3-5]. Third, basically, the chief task of a real-time control program is to deal with all kinds of periodic or aperiodic asynchronous events. However, Java lacks a mechanism for handling these events [6]. Fourth, although Java provides APIs with nanosecond resolution, the real resolution of a clock or a timer is not guaranteed [1].

For developing the standard for real-time Java, IBM, Sun Microsystems, and other organizations from industry and academia formed a team called the *Real-Time for Java Expert Group*, and proposed be called the *Real-Time Specification for Java (RTSJ)* [7, 8]. RTSJ is the definitive reference for the semantics, extensions, and modifications to the Java programming language that enable the Java platform to meet the requirements and constraints of real-time system predictability, performance, and capabilities. This specification provides programmers with the ability to model applications and program logic that require predictable execution, which meets hard real-time constraints. However, the development of the RTSJ-compliant Java Virtual Machine has been slow for most vendors of real-time operating systems so that we cannot use an RTSJ-compliant JVM in the project engaged by the authors. Consequently, we decided to design and implement a real-time extension library that can satisfy the basic requirement of developing a real-time control program.

The real-time extension library has the following design goals: First, the implementation of this extension library should not modify the underlying Java virtual machine. This is because the source code of the JVM running on top of the real-time operating system that we use is not available. Second, we hope that the control program developed with this extension library can be moved to a forthcoming RTSJ-compliant Java virtual machine with a only little modification. Hence, the function and application interface of the extension library should be as similar to the ones of the RTSJ standard as possible.

After analyzing the previously mentioned problems of thread scheduling and garbage collection, we find they are not as severe as we first thought. If a priority-based real-time operating system and a non-RTSJ Java virtual machine that offers kernel-level Java threads are employed, the scheduling semantics of the non-RTSJ Java virtual machine would be very similar to the one of the RTSJ JVM. For solving the problem induced by garbage collection, in addition to heap, RTSJ defines new kinds of memories, and they are scoped memory and immortal memory. Just like heap, Java objects can be created on top of these new memories, but the life cycle of these objects is different from the life cycle of objects created in heap. If an RTSJ real-time Java thread uses only the Java objects in scoped memory or in immortal memory, it can be prevented from the disturbance of the heap garbage collection. However, since it is almost impossible to provide the RTSJ new memory models without modifying the underlying Java virtual machine, here we choose another simpler way to prevent the problem incurred by garbage

collection. After the initialization phase of a real-time control program, the garbage collector of the Java virtual machine will be closed. Heap allocation can happen only in the program initialization phase, and the real-time control program should recycle the pre-allocated objects by itself after initialization [9, 10]. This heap objects recycle approach of our real-time extension library can be easily ported to an RTSJ-compliant JVM if the following replacement is performed. Each real-time Java thread in the original non-RTSJ JVM should be replaced by an instance of the `NoHeapRealtimeThread` class of the RTSJ. Besides, the heap is replaced by the RTSJ immortal memory for object sharing and recycling among real-time Java threads.

In brief, the purpose of our extension library is to solve the remaining two problems stated in the beginning of this section, and it offers the following three mechanisms:

- An asynchronous event handling mechanism that is derived from the one in the RTSJ standard.
- A real-time clock that is also derived from the one in the RTSJ standard. Its resolution is equal to the timer supported by the underlying real-time operating system.
- A Java class that wraps the clock cycle counter of a CPU. The resolution of a clock cycle counter equals the clock cycle time of the CPU. This can improve the resolution of measuring the elapsed time of fine-grained activities on Java.

The remainder of this paper is organized as the follows. The architecture of the real-time extension library is shown in section 2. Section 3 describes both the Java interface and the Java classes which constitute the application interface of the real-time extension library. Section 4 presents the implementation of both event sources and asynchronous event handlers. Section 5 is the experiment section that exploits the significance of the indetermination of timing this real-time extension library. Finally, section 6 concludes the paper.

## 2. THE SYSTEM ARCHITECTURE

This section explains the system architecture, shown in Fig. 1, of the portable real-time extension library. Here the real-time operating system is QNX Neutrino 2.1 [11]. It is compatible with the POSIX real-time standards [12, 13]. The Java virtual machine is the J9 JVM bundled with the IBM VisualAge Micro Edition 1.1. As shown in Fig. 1, our real-time extension library has two parts: the Java part and the native code part. The former contains the Java classes that can interact directly or indirectly with the Java real-time control program. These classes can be subdivided into three kinds, as following:

- The classes relating to asynchronous event handlers.
- Asynchronous event sources including timer, POSIX signal, hardware interrupt, and user-defined asynchronous events.
- The classes relating to the definition of time and real-time clock.

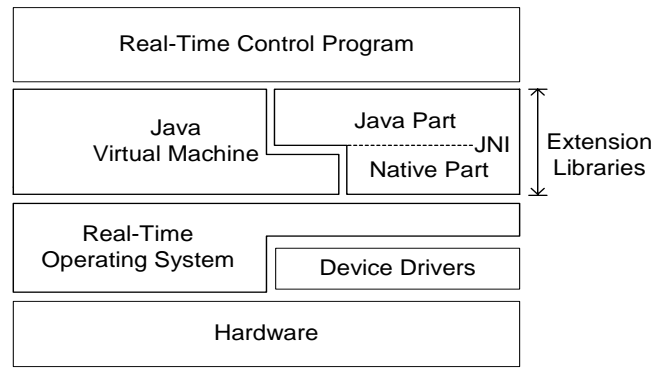


Fig. 1. The architecture of the real-time extension library.

Except for the classes relating to asynchronous event handlers and user-defined asynchronous events, all other Java classes have one or more native methods that deal with the resources offered by the Neutrino real-time operating system. These native methods interact with the Java part classes through the Java native interface (JNI). These interactions can be divided into the following two categories:

- Java classes invoke native methods. Several Neutrino system calls have to be invoked when a Java program sets the resources provided by the Neutrino, for example, to acquire a POSIX real-time timer, to set a POSIX signal handler, or to hook a interrupt service routine, etc.
- A programmer-defined Java asynchronous event handler has to be triggered through the JNI callback function when the native method intercepts a corresponding asynchronous event that happened outside the real-time control program.

Basically, the extension library can be easily ported to another real-time operating system when the following capabilities are available:

- Its Java virtual machine provides kernel-level Java threads and the Java native interface.
- It employs at least a fixed-priority, preemptive scheduling algorithm, and provides the mechanisms that are similar to the POSIX.1b [12] real-time clocks and real-time timers.
- Non-blocking messages can be delivered from its signal handlers and interrupt service routines.
- The CPU has a built-in clock cycle counter.

The first three conditions mentioned above can be satisfied by most mainstream real-time operating systems, such as LynxOS [10, 14] and VxWorks [15, 16]. Furthermore, for the Intel 80X86 family of processors, the built-in processor clock counter was first introduced in the Pentium processor [17-19]. Hence, if a PC-based controller is used, in most cases, the processor inside it will contain a processor clock cycle counter.

### 3. APPLICATION INTERFACE

Fig. 2 shows both the Java interface and the Java classes which constitute the application interface of the real-time extension library. All of them belong to the `javax.realtime` package. They can be divided into the following four main categories:

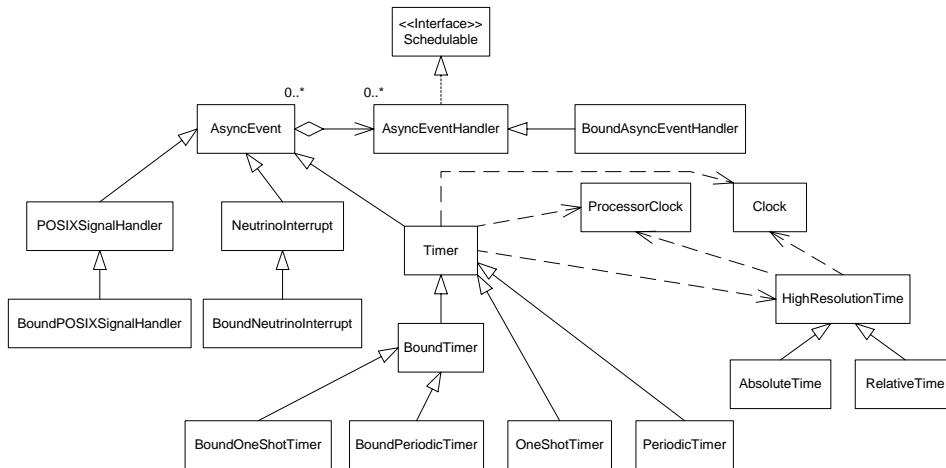


Fig. 2. The Java interface and the Java classes that constitute the application interface of the real-time extension library.

- User-defined asynchronous events.
- External asynchronous events, such as timers, POSIX signals, and hardware interrupts.
- Asynchronous event handlers.
- Java classes that define the notion of time, the processor clock cycle counter, and the real-time clock.

In fact, the RTSJ standard defines only the Java classes that denote timers and POSIX signals. The Java classes for hardware interrupts are our own extensions.

The `AsyncEvent` class is for user-defined asynchronous events. Each instance of this class represents a user-defined event. It can associate more than one asynchronous event handler, which will be executed after the user-defined event occurs. In addition, this class is also the super class of other Java classes that represent external asynchronous events. There are two kinds of Java classes for an external asynchronous event. The first is the Java class whose name has a “Bound” prefix. Each instance of this kind of class has a dedicated notification thread. A notification thread itself is a Java thread. After it is created, it goes through the Java native interface to the native code part of the real-time extension library, and waits inside a message loop to accept the asynchronous messages generated by external event sources. Once an asynchronous message is received, it will use the JNI callback to trigger the asynchronous event handlers associated with the corresponding instance of a Java class that represents an external asynchronous

event. Basically, this kind of Java class requires more operating system kernel resources. However, due to the dedicated notification thread, more CPU time is allocated to deliver the triggered external event, and the average latency of event triggering will become shorter. In fact, these Java classes with dedicated notification threads do not exist in the RTSJ standard, and they are also our own extensions. The second kind of Java class that represents an external asynchronous event uses a shared notification thread. These classes are the original RTSJ Java classes for external asynchronous events. All instances of a Java class that belong to this kind share a notification thread. This kind of implementation prefers the kernel resources to the delay of event triggering.

The real-time extension library also offers two kinds of RTSJ-style asynchronous event handlers. The first is the `AsyncEventHandler` class. Each instance of this class represents an asynchronous event handler. When an asynchronous event handler is triggered to run, its execution is started from the `run()` method of the instance of `AsyncEventHandler`. This is quite similar to the start-up of a Java thread. However, in the RTSJ standard, it does not enforce that each instance of `AsyncEventHandler` have its own Java thread to execute the body of the event handler. In fact, the instances of `AsyncEventHandler` that have been triggered can be scheduled by a scheduler using Java threads. This issue will be discussed in the next section. The second kind of asynchronous event handler is the `BoundAsyncEventHandler` class, and is a subclass of the `AsyncEventHandler` class. However, each instance of this class has a dedicated Java thread that will execute the body of the asynchronous event handler when it is triggered. This implementation improves the turnaround time of an asynchronous event handler by consuming more kernel resources.

At the end of this section, we give a glance at the remains of the Java classes in Fig. 2. The `HighResolutionTime` class, the `AbsoluteTime` class, and the `RelativeTime` class define the notion of time in the RTSJ standard. The `Clock` class implements the real-time clock in the RTSJ standard. The `ProcessorClock` class is our extension for wrapping the clock cycle counter of the Intel 80X86 family of processors (called time-stamping counter by Intel) in a Java class.

#### 4. IMPLEMENTATION

In this section, we first describe the interactions between event sources and asynchronous event handlers. Then, we show the implementations of both. Fig. 3 shows the sequence of actions when a user-defined event is triggered. Once the `fire()` method of `AsyncEvent` is invoked, it first calls the `fire()` method of each associated instance of `BoundAsyncEventHandler`. This will wake up the dedicated kernel-level Java thread of each above instance to execute the body of the asynchronous event handler. Then, the `fire()` method of `AsyncEvent` calls the `fire()` method of `AsyncEventHandlerScheduler`, and passes the instances of the associated instances of `AsyncEventHandler` as parameters of the method invocation. This will cause these instances of `AsyncEventHandler` to be scheduled to run.

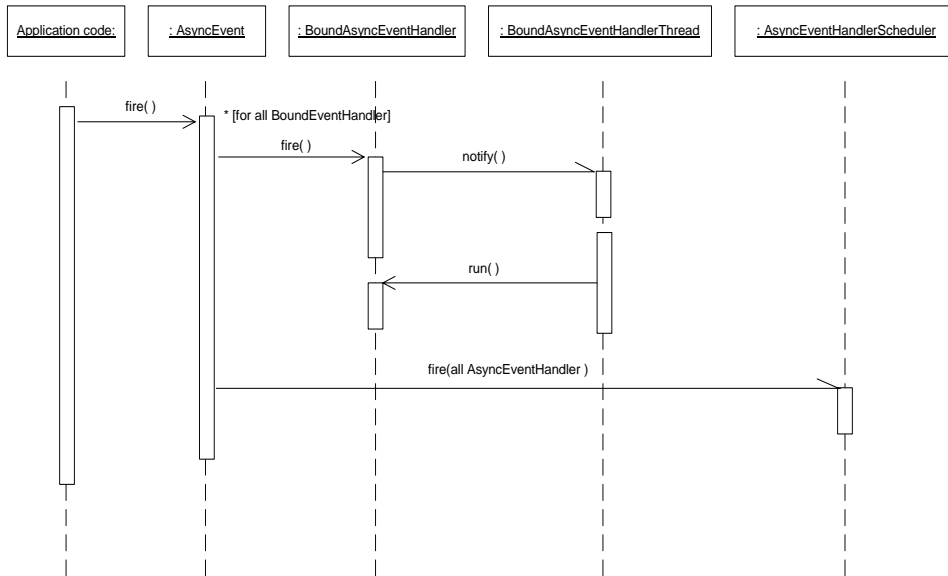


Fig. 3. The sequence diagram of triggering an asynchronous event.

Fig. 4 shows the class hierarchy of the implementation of the two kinds of asynchronous event handlers. Two classes, the `ThreadPool` and `AsyncEventHandlerScheduler`, implement the scheduler for the triggered instances of `AsyncEventHandler`. We will present the detail about this scheduler in the following subsection. The scheduler for another kind of asynchronous event handler, `BoundAsyncEventHandler`, is implemented as a method embedded inside itself. `BoundAsyncEventHandlerThread` implements the dedicated kernel-level Java thread of each instance of `BoundAsyncEventHandler`.

#### 4.1 The Scheduler for AsyncEventHandler Class

Each instance of the `AsyncEventHandler` class, like a Java thread, has a priority for scheduling. Moreover, the range of priority of the `AsyncEventHandler` class is the same as the range of priority of a Java thread. In the IBM J9 Java virtual machine, the range of priority is from 1 to 10. In fact, RTSJ enforces that an RTSJ-compliant JVM should at least implement a fixed-priority, preemptive scheduling algorithm with at least 28 unique priority levels. However, after analyzing the requirement of our application—the real-time control program inside a plastic-injection-molding machine, we find that it does not need so many priority levels. For simplifying the implementation of the real-time extension library, we decide to use the range of priorities of a kernel-level Java thread in the non-RTSJ JVM as the range of priorities of the `AsyncEventHandler` class. This design decision will not complicate the task for porting our real-time application to the forthcoming RTSJ-compliant Java virtual machine.

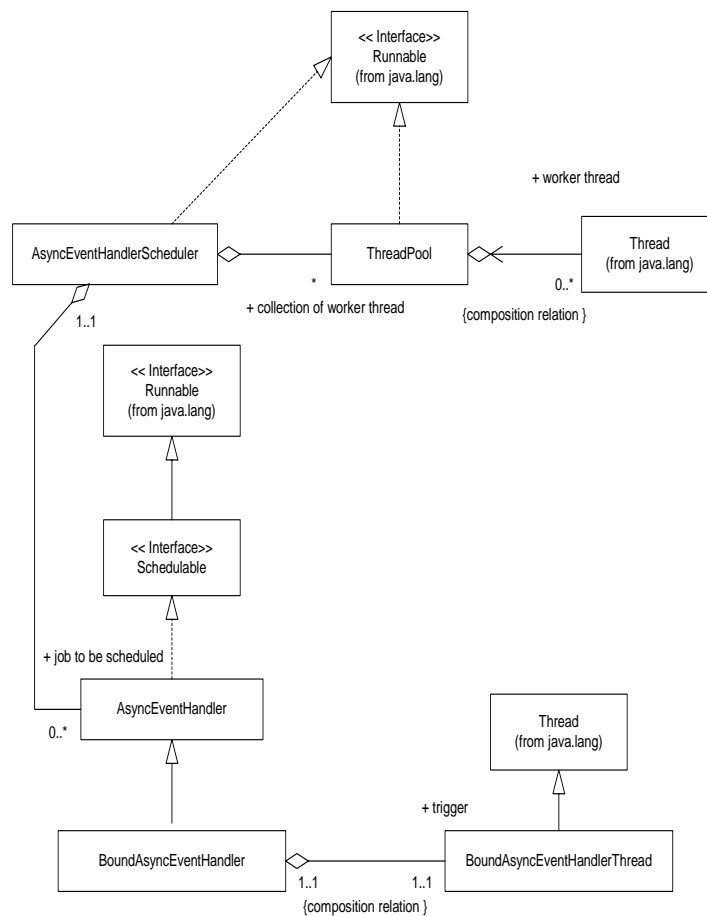


Fig. 4. The class hierarchy of the implementation of asynchronous event handlers.

Fig. 5 shows the architecture of the scheduler for the triggered instances of `AsyncEventHandler`. This scheduler has ten FIFO ready queues, and each of them has a different priority for scheduling. When an instance of `AsyncEventHandler` is triggered, it will be appended to the end of one of the above ready queues according to its priority. This scheduler use pre-allocated Java threads as worker threads to execute the triggered instances of `AsyncEventHandler`. A worker thread has three states: *active*, *bound*, and *inactive*. Each FIFO ready queue of the scheduler has a dedicated local worker thread pool, and it stores inactive worker threads that has the same priority as the FIFO ready queue. In addition to the local worker thread pools, this scheduler also has a global worker thread pool. However, the inactive worker threads inside it may have different priorities.

For each FIFO ready queue that contains any triggered instances of the `AsyncEventHandler` class, an active worker thread is allocated from either the local or the global worker thread pool to execute these triggered instances sequentially. The priority of an active worker thread is equal to the priority of the FIFO ready queue it

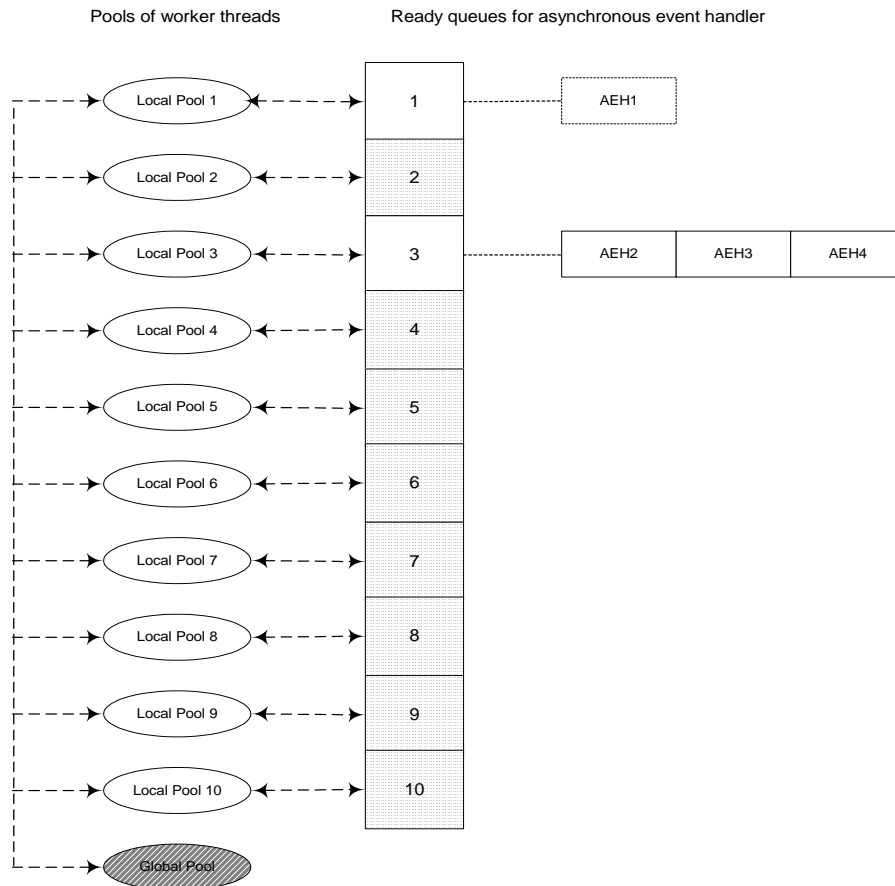


Fig. 5. The architecture of the scheduler of the AsyncEventHandler class.

serves. Since a thread allocated from the global worker thread pool requires an extra OS kernel operation to change its priority, the local thread pool is prior to the global one for allocating an active worker thread. Because each FIFO ready queue has its own active worker thread (kernel-level Java thread), for two instances of the AsyncEventHandler class that have different priorities, the scheduling among them is preemptive.

The RTSJ standard allows an asynchronous event handler to contain blocking operations. Hence, the scheduler in Fig. 5 also has to know how to deal with this situation. As shown in Fig. 5, suppose the instance **AEH2** has a blocking operation. In order to prevent the following **AEH3** and **AEH4** from being blocked, before the **AEH2** enters the blocking operation, it has to call the `AsyncEventHandler.bindToDedicatedThread()` method. Then, this method will invoke the `AsyncEventHandlerScheduler.bindToDedicatedThread()` method to switch the current active worker thread to a bound worker thread. The bound worker thread will be dedicated to the **AEH2** until the body of **AEH2** is finished. Once the active worker thread mentioned above becomes a bound worker thread, a new active worker thread is immediately allo-

cated from either the local or the global worker thread pool to run the following **AEH3** and **AEH4**.

For using this scheduler, the size and the initial number of inactive worker threads of each worker thread pool have to be specified during the initialization of a real-time control program. However, because the types and the patterns of asynchronous events that will be handled by the real-time control program are predictable, we assume that the previously mentioned approach is feasible.

#### 4.2 BoundAsyncEventHandler Implementation

Fig. 6 shows the core code of BoundAsyncEventHandler and the BoundAsyncEventHandlerThread. When a new instance of BoundAsyncEventHandler is created, its internal dedicated worker thread, an instance of BoundAsyncEventHandlerThread, is also created. This class is a subclass of `java.lang.Thread`. The dedicated worker thread will enter the dispatching loop of BoundAsyncEventHandler (the `schedule()` method) after the new instance of BoundAsyncEventHandler is initialized. When an asynchronous event has arisen, the corresponding instance of AsyncEvent will call the `fire()` method of each instance of BoundAsyncEventHandler attached with the instance of AsyncEvent. The `fire()` method will increase the internal variable `FireCount` of the instance of BoundAsyncEventHandler. The worker thread executes the `run()` of BoundAsyncEventHandler if the `FireCount` variable is larger than 0. The `run()` method can be overridden by programmers, and it can modify the value of the `FireCount` by calling `getAndDecrementPendingFireCount()` (to decrease the `FireCount` by one) or `getAndClearPendingFireCount()` (to clear the `FireCount` to zero). After `run()` is finished, the worker thread executes the body of the asynchronous event handler, `handleAsyncEvent()` of BoundAsyncEventHandler.

```

public abstract class BoundAsyncEventHandler
    extends AsyncEventHandler {
    BoundAsyncEventHandlerThread HandlerThread;
    public AsyncEventHandler(int Priority) {
        super(Priority);
        HandlerThread = new
            BoundAsyncEventHandlerThread(this);
        HandlerThread.start();
    }
    void scheduler() {
        while(True) {
            synchronized(this) {
                while(FireCount == 0)
                    Wait();
            }
        }
    }
}

```

```

        run();
    }
}
public synchronized void fire() {
    FireCount++;
    if(FireCount == 1)
        Notify();
}
}

class BoundAsyncEventHandlerThread extends Thread {
    BoundAsyncEventHandler Handler;
    BoundAsyncEventHandlerThread(BoundAsyncEventHandler Handler) {
        this.Handler = Handler;
    }
    run() {
        Thread.currentThread.setPriority(Handler.Priority);
        Handler.scheduler();
    }
}
}

```

Fig. 6. Core code of the BoundAsyncEventHandler and BoundAsyncEventHandlerThread class.

### 4.3 External Asynchronous Event

There are three kinds of external asynchronous events: timer, POSIX signal, and hardware interrupt. For each kind of event, two implementations of Java class are offered. One uses a dedicated notification thread, and the other uses a shared notification thread. Consequently, there are a total of six combinations of the Java classes for external asynchronous events. However, since the core implementations of these Java classes are similar, here we only show the detail of using a shared notification thread to implement timers.

There are two kinds of timers—OneShotTimer (the timer of its instance expires only once) and PeriodicTimer (the timer of its instance expires periodically). The common super class of these two classes is the Timer class. All instances of the two kinds of timer classes share the same notification thread, which was implemented by the Timer class. Each instance of the Timer class has a corresponding Neutrino timer in the native code part of the extension library. As shown in Fig. 7, in the native code part of the extension library, the states of each instance of the Timer class are stored in five arrays, and each Timer class instance has a unique array index. The states stored in the above-mentioned five arrays are:

- The JNI global object reference of each instance of the Timer class.
- The ID of each Neutrino timer.
- A Boolean state representing the presence of asynchronous event handlers for an instance of the Timer class.

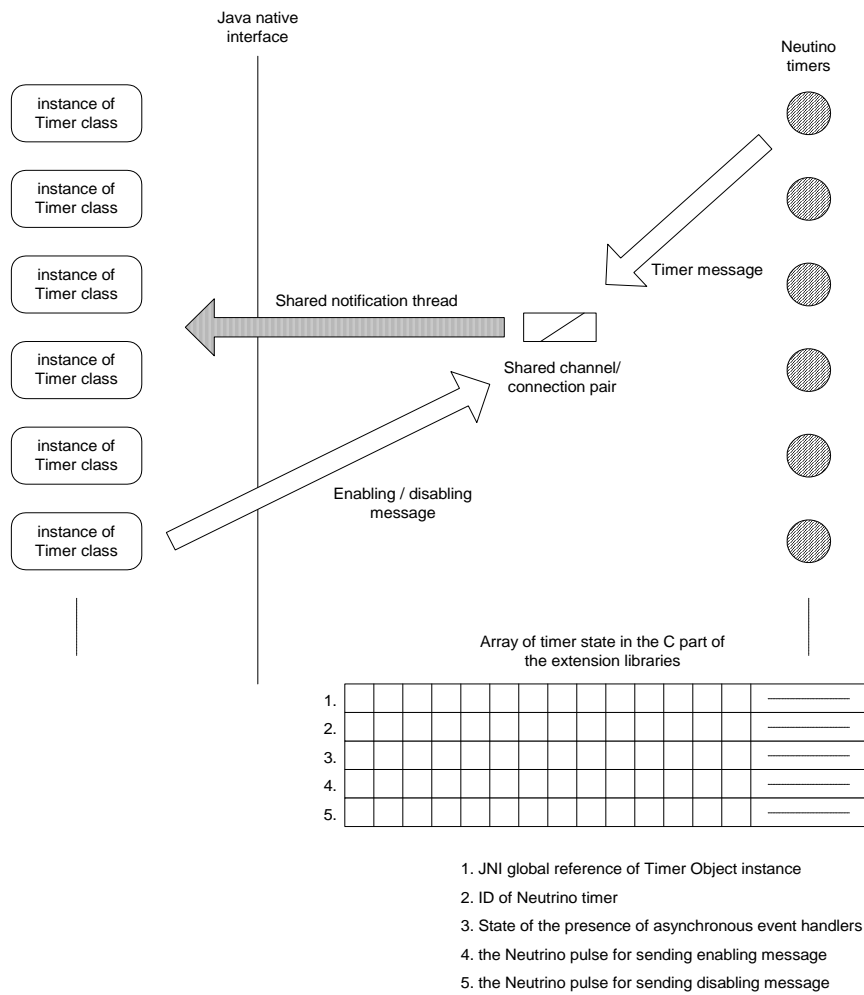


Fig. 7. The implementation of timer.

- The Neutrino pulse for sending enabling messages.
- The Neutrino pulse for sending disabling messages.

*Pulse* is a mechanism for delivering asynchronous messages in the Neutrino real-time operating system. The message receiver receives the messages from a message queue called the *channel*. A delivered asynchronous message contains two parts, a code part and a data part. Three kinds of asynchronous messages are delivered to the notification thread. The first is the enabling message. The code part of this kind of message is `ENABLING_MESSAGE`, and the data part is the array index of the `Timer` instance that delivers this message. If a new asynchronous event handler is attached to an instance of the `Timer` class that originally contains no handler, the instance of `Timer` class will deliver an enabling message to the notification thread. The second kind of message is the disabling message, whose format is quite similar to the enabling

message, except the code part becomes `DISABLING_MESSAGE`. This kind of message is output when the last asynchronous event handler of an instance of the `Timer` class is removed. Fig. 8 shows the message loop of the shared notification thread in the `Timer` class. After the notification thread receives an enabling message or a disabling message, it will change the state of handler presence according to the array index retrieved from the data part of the received message.

The third kind of message is the timer message, which is delivered when a Neutrino timer expires. The code part of this kind of message is `TIMER_MESSAGE`, and the data part is also the array index of the expired Neutrino timer. As shown in Fig. 8, after the notification thread receives a timer message, it checks whether or not the corresponding instance of the `Timer` class contains asynchronous event handlers. If any asynchronous event handler is present, the notification thread will invoke the `fire()` method of the corresponding instance of the `Timer` class through the JNI callback. JNI can reach the `fire()` method by passing both the JNI global reference of the target instance of the `Timer` class (`timer_instance[message.data]`) and the JNI method ID of the `fire()` method (`fire_method_ID`). Finally, the associated asynchronous event handlers will be scheduled to run.

The implementation of the POSIX signal is similar to the implementation of the timer. However, because the POSIX signal is delivered to a thread, we have to use two numbers (Neutrino thread ID, signal number) as the indexes for accessing the states of each instance of the `POSIXSignalHandler` class inside the native part of the extension library. Because the actually used indexes are sparse in comparison to their range, we decide to use a hash table to store the states of each instance of the `POSIXSignalHandler` class, and the hash key is (Neutrino thread ID, signal number).

```

while( 1 ) {
    MessageReceive(channel_ID, &message, sizeof(message));
    switch(message.code) {
        case TIMER_MESSAGE:
            if(presence_status[message.data] == ENABLE)
                JNI_callback(timer_instance[message.data], fire_method_ID);
            break;
        case ENABLING_MESSAGE:
            presence_status[message.data] = ENABLE;
            break;
        case DISABLING_MESSAGE:
            presence_status[message.data] = DISABLE;
            break;
    }
}

```

Fig. 8. The message loop of the shared notification thread in the `Timer` class.

The RTSJ standard treats an asynchronous event handler as a real-time thread. We assume that the conventional interrupt service routines (ISRs) cannot be entirely replaced

by the asynchronous event handlers. Since an ISR can interrupt the execution of any running threads, its length is usually very short. However, the overhead of scheduling an asynchronous event handler to run is too high to replace a conventional ISR. We think that the best way to integrate hardware interrupts into our extension real-time library is to combine ISRs with the use of asynchronous event handlers. ISRs are responsible for dealing with time-critical and device-dependent tasks, while asynchronous event handlers are responsible for dealing with time-insensitive. Our real-time extension library offers the APIs an ISR to trigger the asynchronous event handlers if necessary. This combination of ISR and asynchronous event handlers can make the tasks of processing interrupts more modularized.

At the end of this subsection, we briefly mention the implementation of the Java external event classes that have dedicated notification threads. Fig. 9 shows the message loop of `BoundTimer`. Since each instance of these has a dedicated notification thread, the native part states of each object instance can be stored in the stack of the dedicated notification thread. In other words, these states are defined as automatic variables inside the C language function that contains the message loop. The fore mentioned centralized data structures such as arrays or hash tables are no longer required.

```

while( 1 ) {
    MessageReceive(channel_ID, &message, sizeof(message));
    switch(message.code) {
        case TIMER_MESSAGE:
            if(presence_status == ENABLE)
                JNI_callback(timer_instance, fire_method_ID);
            break;
        case ENABLING_MESSAGE:
            presence_status = ENABLE;
            break;
        case DISABLING_MESSAGE:
            presence_status = DISABLE;
            break;
    }
}

```

Fig. 9. The message loop of the dedicated notification thread in the `BoundTimer` class.

## 5. THE INFLUENCE ON TIMING

In this section, we will try to illustrate how our extension library impacts on timing by comparing the precision of the Neutrino periodic timer with the precision of the periodic timer of our extension library. For this purpose, the following experiment is performed with both kinds of periodic timers. First, we create a new periodic timer, and set its period to 20 ms. Then, we observe the expiration of this timer for 1000 times, and record the interval between each pair of consecutive expirations. In the Neutrino-version experiment, we use its POSIX thread and POSIX timer to implement the experiment

stated above. In the Java-version experiment, two object instances of the following two classes in our extension library are employed: `BoundAsyncEventHandler` and `BoundedPeriodicTimer`. Both versions of the experiment are performed on a PC-based controller with a 550 MHz Intel Celeron processor. The operating system is QNX Neutrino 2.1, and the Java virtual machine is the J9 JVM of the IBM VisualAge Micro Edition 1.1.

In order to simulate the results of the activities that may happen in the real world, while the experiment is running, we also select one of the following six workloads and execute it. For preventing the side effects of a finished experiment affect the result of the next experiment, we reboot the test platform before starting each experiment.

- **No load:** Neither foreground programs nor background processes are running.
- **Background load:** No foreground program is running. All background processes are removed except the following three daemon processes, `slinger`, `inetd`, and `syslogd`.
- **Full load:** Two workloads, hard disk load and network load, are running concurrently.
- **Hard disk load:** Running the shell command “`ls -lR / &`”. Because the test platform is rebooted before performing an experiment, the disk cache will not hold the disk blocks of file directories that are touched in the previous experiment. This is important since the disk blocks touched by the “`ls -lR / &`” command can almost be accommodated by the disk cache.
- **Network load:** Using the “`ping -f`” command repeatedly to send ICMP (Internet Control Message Protocol) packages to another machine on the same LAN.
- **Calculation load:** A C program that performs floating-point division inside an infinite loop. Here we carefully turn off the compiler optimization option to make sure that the compiler will not eliminate the floating-point division instructions.

Notice that the priorities of threads that execute foreground workloads mentioned above (full load, hard disk load, network load, and calculation load) are always lower than those of the threads (either the POSIX thread or the kernel-level Java thread of the instance of `BoundAsyncEventHandler`) that handle the events of timer expiration.

In the Neutrino-version experiment, the expiration time of the Neutrino timer is measured by reading the time-stamping counter of the Celeron processor through inline assembly code. In the Java-version experiment, `ProcessorClock` is employed in the instance of `BoundAsyncEventHandler` to measure the expiration time of the instance of `BoundedPeriodicTimer`. However, since `ProcessorClock` has to go through the Java native interface to read the time-stamping counter, for controlling the accuracy of the experiment, we have to understand the timing behavior of the JNI. As shown in Table 1, the overhead of JNI is stable, and the six kinds of workloads mentioned previously almost have no influence on it (the ranges of variation, **Max–Min** of each row in Table 1, are all within 49 nanoseconds). In addition, the JNI overhead (about 1 microsecond) is quite small in comparison to the range of variation of a periodic timer’s interval (about 1 to 2 milliseconds, shown in Tables 2 and 3). Hence, we think the influence of JNI can be omitted in the experiment.

Fig. 10 shows the distribution of expiration time of periodic timers under six kinds

of workloads. We can find that the Neutrino-version experiment and the Java-version experiment have similar distributions. The distribution of the Neutrino-version experiment is more concentrate than the distribution of the Java-version. Tables 2 and 3 summarize the experimental results of both kinds of periodic timer. The range of variation of the Neutrino periodic timer (**Max–Min** of each row in Table 2) is from 1003.3 to 1208.2 microseconds. The range of variation of the periodic timer of our real-time extension library is from 1153.6 to 2289.8 microseconds. Hence, in the worst case, our real-time extension library introduces about one more millisecond indetermination of timing than the Neutrino real-time operating system. However, such resolution of timing is enough for our application-the real-time control program inside plastic injection molding machine. In such a machine, the maximum tolerable delay of a control command is about 6 milliseconds.

**Table 1. The overhead of JNI influenced by other workload (ns).**

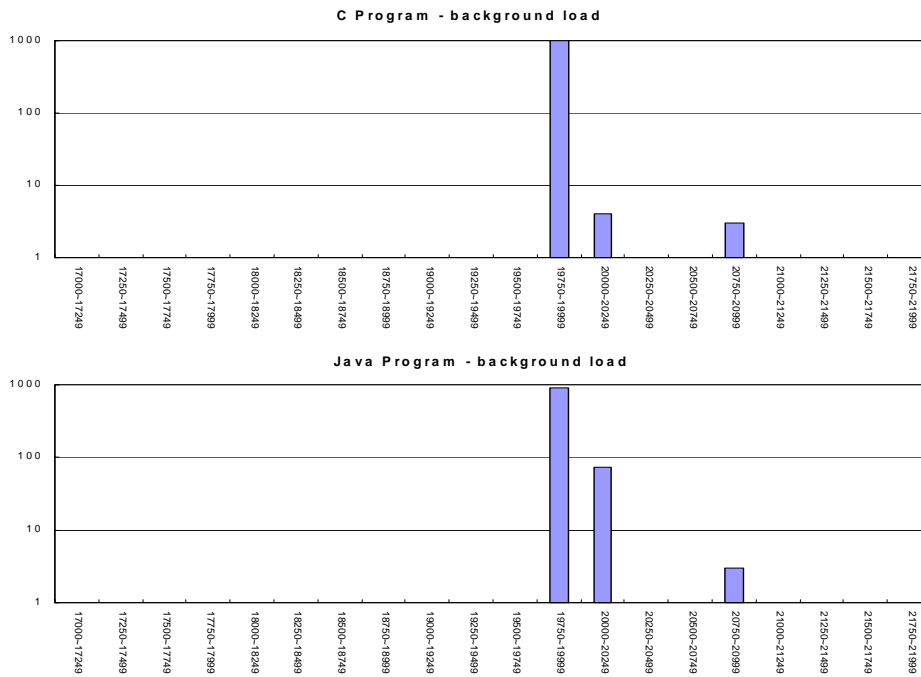
	Min	Max	Range of Var.	Avg.	Std. Dev.
No load	1025	1065	40	1025	1
Background load	1025	1065	40	1025	2
Full load	1025	1067	42	1025	2
Hard disk load	1025	1068	43	1025	2
Network load	1025	1065	40	1025	1
Calculation load	1025	1074	49	1025	2

**Table 2. The experiment results of the Neutrino periodic timer ( $\mu$ s).**

	Min	Max	Range of Var.	Avg.	Std. Dev.
No load	19987.9	20992.9	1005	19996.2	3.7
Background load	19978.9	20997.9	1019	19996.8	3.4
Full load	19844.8	20992.8	1148	19996.2	7.9
Hard disk load	19977.9	20993	1015.1	19996.5	5.3
Network load	19840.8	21049	1208.2	19996.2	10.4
Calculation load	19991	20994.3	1003.3	19996.4	2.5

**Table 3. The experiment results of the periodic timer of our real-time extension library ( $\mu s$ ).**

	Min	Max	Range of Var.	Avg.	Std. Dev.
<b>No load</b>	19835.4	20992.6	1157.2	19995.4	3
<b>Background load</b>	19839.2	20993.5	1154.3	19996.1	4.8
<b>Full load</b>	18849.2	21139	2289.8	19996	8.9
<b>Hard disk load</b>	19839.5	20993.1	1153.6	19996.1	8.5
<b>Network load</b>	19623.4	21052.5	1429.1	19995.9	4.4
<b>Calculation load</b>	19838.5	20993.8	1155.3	19996	3.8



(a) No load

**Fig. 10. The distribution of the expiration time of the periodic timers offered by both the Neutrino and the Java real-time extension library.**

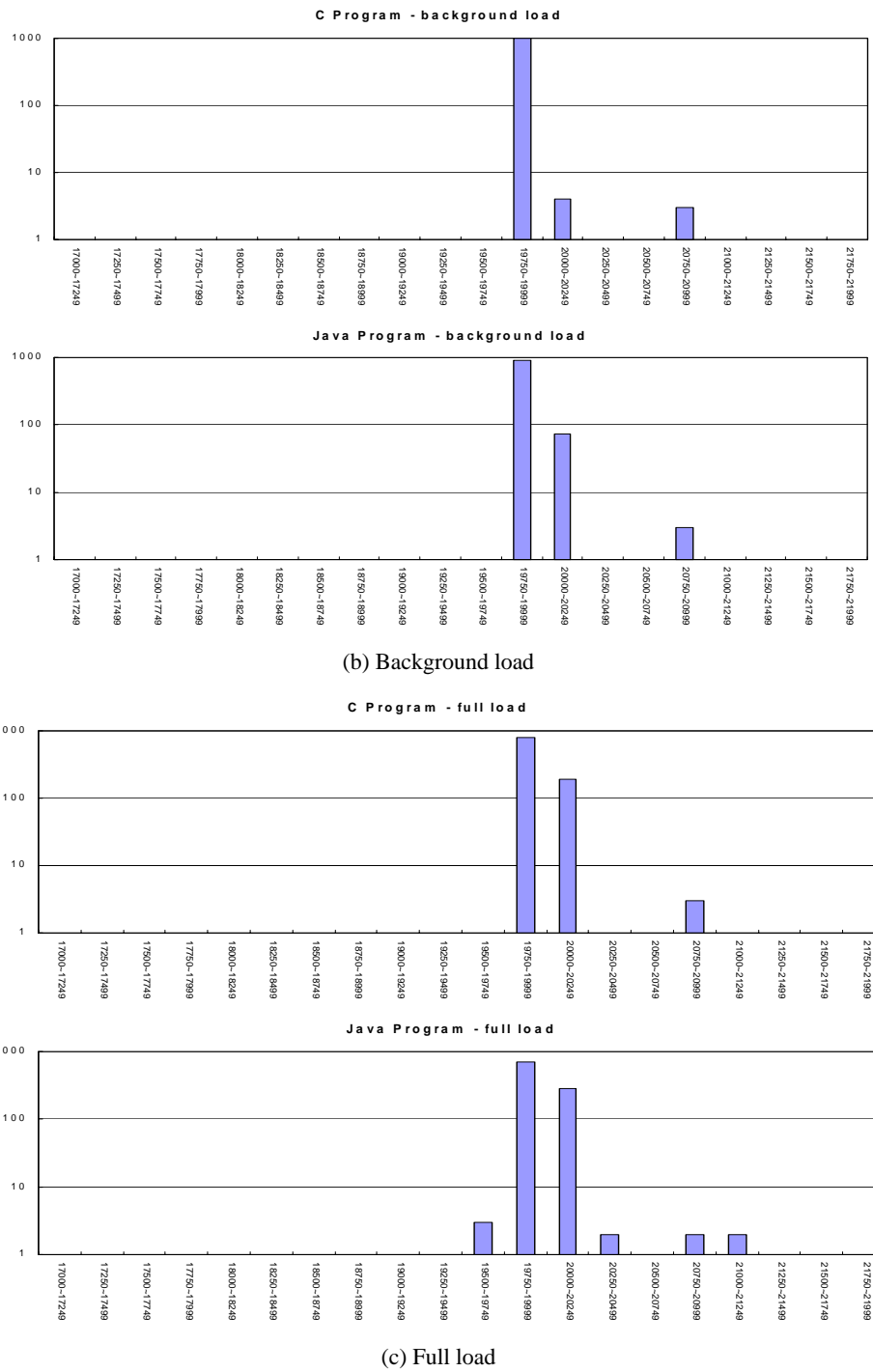
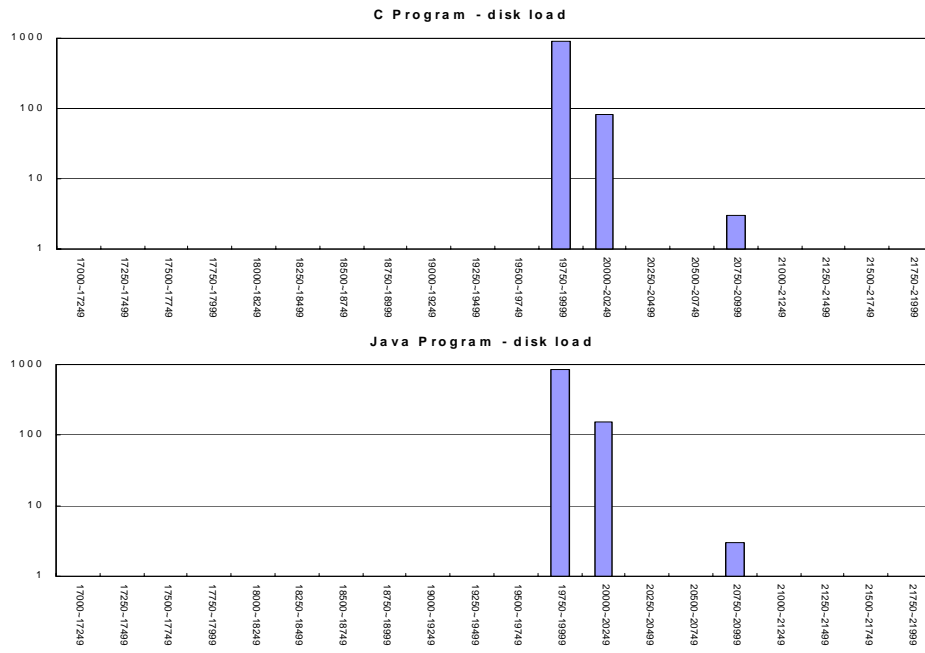
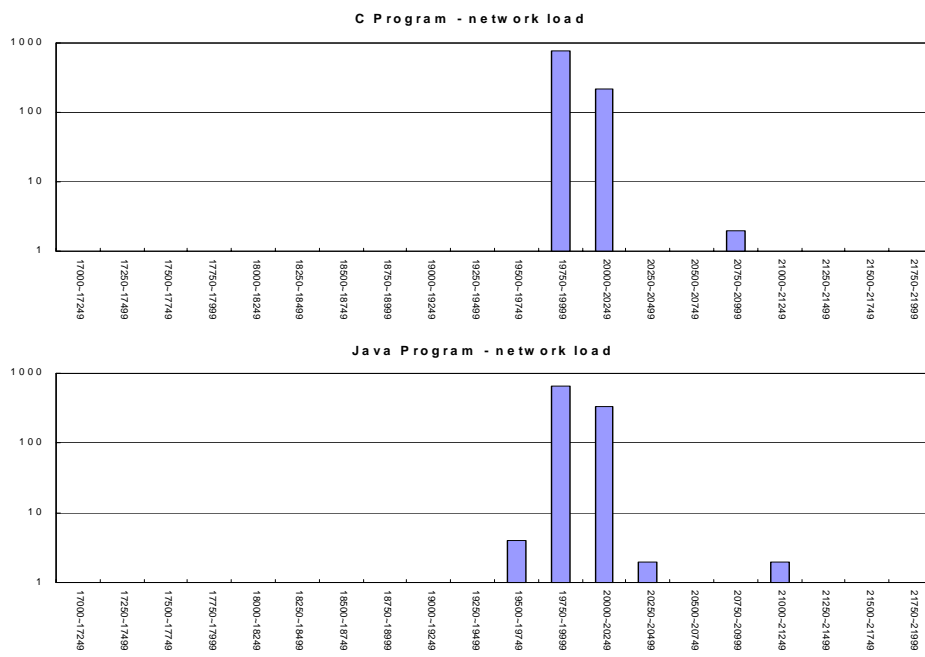


Fig. 10. (Cont'd) The distribution of the expiration time of the periodic timers offered by both the Neutrino and the Java real-time extension library.

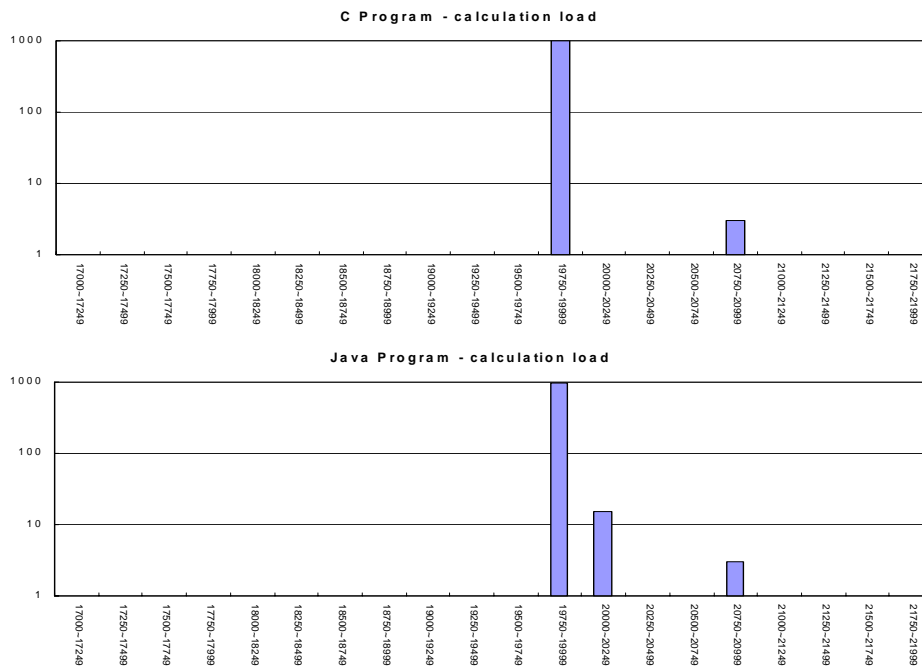


(d) Hard disk load



(e) Network load

Fig. 10. (Cont'd) The distribution of the expiration time of the periodic timers offered by both the Neutrino and the Java real-time extension library.



(f) Calculation load

Fig. 10. (Cont'd) The distribution of the expiration time of the periodic timers offered by both the Neutrino and the Java real-time extension library.

## 6. CONCLUSIONS

For using Java to develop the real-time control program inside an industrial application, we design and implement a real-time extension library to enhance the capability of current Java virtual machine. There are three advantages of this extension library. First, to implement it needs no modification of the underlying Java virtual machine. Second, its implementation is also easy to port to most mainstream real-time operating systems. Third, since this extension library is derived from the RTSJ standard, our real-time control program that is built upon it can be moved to a forthcoming RTSJ-compliant Java virtual machine with little modification. In this paper, we discussed the function, architecture, application-programming interface, and implementation of the real-time extension library. In addition, we also performed several experiments to explore how significant the indetermination of timing this real-time extension library would introduce. From the experiment results, we found that the timing characteristics of this real-time extension library was within the acceptable range of our application.

## REFERENCES

1. B. Brosgol, "A comparison of the concurrency and real-time features of Ada 95 and Java," *Ada Language UK. ADA User Journal*, Vol. 19, 1999, pp. 225-57.
2. K. Nil98, "Adding real-time capabilities to Java," *Communications of the ACM*, Vol. 41, 1998, pp. 49-56.
3. E. Bertolissi and C. Preece, "Java in real-time applications," *IEEE Transactions on Nuclear Science*, Vol. 45, 1998, pp. 1965-1972.
4. W. Foote, "Theory versus practice in real-time computing with the Java platform," in *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999, pp. 105-108.
5. D. Mulchandani, "Java for embedded systems," *IEEE Internet Computing*, Vol. 2, 1998, pp. 30-39.
6. A. Miyoshi, T. Kitayama, and H. Tokuda, "Implementation and evaluation of real-time Java threads," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997, pp. 166-175.
7. The Real Time for Java Expert Group, *The Real-Time Specification for Java*, Addison-Wesley, Reading, Massachusetts, 2000.
8. D. Hardin, "The real-time specification for Java," *Dr. Dobb's Journal*, Vol. 25, pp. 78-84.
9. S. Uckun and F. Gasperoni, "Making Java real-time," *IEEE Spectrum*, Vol. 35, 1998, pp. 22-23.
10. W. Weinberg, "Real-time Java implementation for embedded environments," *Real-Time Magazine*, Vol. 6, 1998, pp. 43-49.
11. QNX Software Systems, *QNX Neutrino System Architecture Guide*, QNX Software Systems, Kanata, Ontario, Canada, 1999.
12. IEEE, *POSIX.1b: System Application Programming Interface—Real Time Extensions*, IEEE Standards Press, Los Alamitos, California, 1993.
13. IEEE, *POSIX.1: System Application Programming Interface—Amendment 2: Thread Extension (C Language)*, IEEE Standards Press, Los Alamitos, California, 1995.
14. Lynx Real-Time Systems, *LynxOS White Paper*, <http://www.lynx.com>, Lynx Real-Time Systems, 1999.
15. Wind River Systems, *VxWorks Programmer's Guide 5.4*, Wind River Systems, Alameda, California, 1999.
16. Wind River Systems, *Personal Jworks 3.02 Datasheet*; [http://www.wrs.com/products/html/persjwks\\_ds.html](http://www.wrs.com/products/html/persjwks_ds.html), Wind River Systems, 2000.
17. B. Brey, *The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor—Architecture, Programming, and Interfacing*, Prentice Hall, Englewood Cliffs, New Jersey, 1997.
18. Intel Corporation, *Intel Architecture Software Developer's Manual—Vol. 3: System Programming Guide*, Intel Corporation, Mount Prospect, Illinois, 1997.
19. T. Shanley, *Pentium Pro and Pentium II System Architecture*, Addison-Wesley, Reading, Massachusetts, 1997.



**Hsin-Ta Chiao (焦信達)** was born on May 29, 1973 in Taichung, Taiwan, Republic of China. He received the BS degree in Computer and Information Science from National Chiao Tung University in 1995. Currently, he is a Ph.D. candidate in Department of Computer and Information Science, National Chiao Tung University. His research interests include real-time systems, distributed systems, and Internet technologies.



**Scott Hsu-Jing Kao (高叔敬)** given birth to in 1977, received his B.S. degree in Management of Information System (MIS) from Kung-Shan University of Science and Technology in 1999, and his M.S. degree in MIS from National Ping-Tung University of Science and Technology in 2001. His research domains include object-oriented analysis/design, computational intelligence, real-time/embedded systems, and mobile agents.



**Yue-Shan Chang (張玉山)** was born on August 4, 1965 in Tainan, Taiwan, Republic of China. He received the B.S. degree in Electronic Technology from National Taiwan Institute of Technology in 1990, the M.S. degree in Electrical Engineering from the National Cheng Kung University in 1992, and the Ph.D. degree from Computer and Information Science at National Chiao Tung University in 2001. Dr. Chang joined the Department of Electronics Engineering of Ming Hsing Institute of Technology as a lecturer in August 1992. Since from August 2001, he became an associate professor. His research interests are in distributed systems, object oriented programming, information retrieval and integration, and internet technologies.



**Shen-Tzay Huang (黃申在)** is currently an associate professor in the department of Management Information Systems at National Pingtung University of Science and Technology. He received his PhD degree in computer science from UCLA in 1993. His current research interests include Java technology, Web engineering, component-based software engineering, business intelligent systems and knowledge management.



**Shyan-Ming Yuan (袁賢銘)** was born on July 11, 1959 in Maui, Taiwan, Republic of China. He received the B.S.E.E degree from National Taiwan University in 1981, the M.S. degree in Computer Science from University of Maryland Baltimore County in 1985, and the Ph.D. degree in Computer Science from University of Maryland College Park in 1989. Dr. Yuan joined the Electronics Research and Service Organization, Industrial Technology Research Institute as a Research Member in Oct. 1989. Since September 1990, he had been an Associate Professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He became a Professor in June, 1995. His current research interests include distributed objects, internet technologies, and software system integration. Dr. Yuan is a member of ACM and IEEE.