Abstract
The Java environment, with its platform neutrality, simplified object model, strong notions of safety and security, as well as multithreading support, provides many advantages for a new generation of networked embedded and real-time systems. However, the large size, nondeterministic behavior, and poor performance of the existing generation of embedded Java-enabled implementations have hampered the acceptance of Java in the real-time and embedded worlds. aJile Systems has pioneered in the development of a hardware based JVM implementation, which makes real-time embedded Java a practical reality. aJile’s microprocessors provide direct support for the entire JVM instruction set and thread model, obviating the need for a Java interpreter or Just-In-Time (JIT) compiler, as well as the traditional Real-Time Operating System (RTOS). aJile’s processor technology also supports multiple JVM contexts executing on the same CPU, enhancing safety and security by guaranteeing space and time allotments for multiple Java applications. Bundled with Sun’s JME/CLDC/MIDP2.0/LWTIWT or CDC/FP/PBP runtime system, optimizing application builder, debugging tools and evaluation systems provides a complete solution for implementing real-time, networked embedded devices, entirely in pure Java technology. The powerful combination of direct JVM bytecode execution, direct multithreading support, and fully protected multiple JVM environments is ideal for efficient, safe, and robust Java execution and dynamic delivery of the Java based applications and services on-demand over Internet.

Introduction
The embedded and real-time system marketplace is exploding in the "post-PC" era, especially as more and more devices are becoming Internet-enabled due to the pervasiveness of the broadband and 3G/4G cellular networks. Networked, real-time embedded systems are becoming common in markets such as telecommunications, industrial automation, home and building control, automotive systems and medical instrumentation. The software content of these networked devices is soaring, putting a significant strain on scarce development resources. The real-time and embedded developer also faces an extremely heterogeneous processing environment, with a plethora of processors, operating systems, and peripheral...
device types. Thus, engineers are increasingly looking to Java technologies to provide a more productive, portable development environment for real-time and embedded systems. These technologies include the Java object-oriented programming language; the Java Virtual Machine; as well as a large selection of runtime class libraries.

**The Java Platform**

The Java platform debuted in 1995 as a desktop and server language environment, although it was originally developed as an object-oriented programming language for embedded devices. The platform neutrality, simplified object model, strong notions of safety and security, as well as multithreading support, of the Java platform provides many advantages for a new generation of networked embedded and real-time systems.

**The Java programming language**

The Java object-oriented programming language [1], with its familiar C-like syntax and simplified object model, is easier to master than C++, and has been shown to be 25% - 40% more productive. Java's lack of user manipulable pointers and automatic memory management are frequently cited as key factors in improved software productivity and robustness. The Java language also supports priority-preemptive multithreading, and the synchronized keyword provides a particularly elegant means of enforcing mutual exclusion synchronization. In fact the Java has become the most popular language for the development of the Internet based applications as illustrated in the below figure.
Java execution is supported by a standardized Java Virtual Machine (JVM) to which Java programs are compiled [7]. This design enables Java to be highly portable since Java programs can run on any system that supports the JVM. Moreover, the underlying target computer system implementation is insulated from the application by the JVM, thus enhancing the safety and security of mobile code. Indeed, the entire Java environment is designed to support network-distributed computing. The JVM “bytecode” instruction set is a stack-based 32-bit architecture, with a number of unique instructions, such as virtual method invocation with lock detection. In a typical Java runtime environment, portable applications or applets compiled into Java Virtual Machine bytecodes are interpreted by a software JVM implementation, Just-In-Time (JIT) compiled to native machine code, or directly executed by a Java microprocessor. A Java microprocessor is the most space- and time-efficient vehicle for executing Java bytecode.

*Figure 1.* Java programming language

**The Java Virtual Machine (JVM)**

Java execution is supported by a standardized Java Virtual Machine (JVM) to which Java programs are compiled [7]. This design enables Java to be highly portable since Java programs can run on any system that supports the JVM. Moreover, the underlying target computer system implementation is insulated from the application by the JVM, thus enhancing the safety and security of mobile code. Indeed, the entire Java environment is designed to support network-distributed computing. The JVM “bytecode” instruction set is a stack-based 32-bit architecture, with a number of unique instructions, such as virtual method invocation with lock detection. In a typical Java runtime environment, portable applications or applets compiled into Java Virtual Machine bytecodes are interpreted by a software JVM implementation, Just-In-Time (JIT) compiled to native machine code, or directly executed by a Java microprocessor. A Java microprocessor is the most space- and time-efficient vehicle for executing Java bytecode.
Interpreters are slow (up to an order of magnitude slowdown) and require memory to store the interpreter code. JIT’s require sizable memory areas to hold the translated code, and execution in a JIT environment suffers from cache miss effects if control transfers to a non-translated code block. However, building such a processor is a nontrivial task, as the JVM instruction set is quite complex, much more so than even traditional Complex Instruction Set Computers (CISC’s) such as the Intel x86, Motorola 680x0, and ARM families.

Java Micro Edition (JME)
Java Platform, Micro Edition (Java ME) is a collection of technologies and specifications to create a platform that fits the requirements for mobile devices, consumer devices, and embedded devices. It was originally created in order to deal with the constraints associated with building applications for small devices. For this purpose Sun defined the basic Java ME technology to fit such a limited environment and make it possible to create Java applications running on small devices with limited memory, display and processor power capability. It provides a robust, flexible environment for applications running on mobile and other embedded devices such as mobile phones, personal digital assistants (PDAs), TV set-top boxes, printers, thin clients, and M2M network edge appliances. Java ME includes flexible user interfaces, robust security, built-in network protocols, and support for networked and offline applications that can be downloaded dynamically. Applications based on Java ME are portable across many devices, yet leverage each device’s native capabilities.

The Java ME Platform is a truly open solution for building mobile applications. It comprises a number of specified components, which have been defined by the industry through the Java Community Process (JCP). Java ME is designed to provide portability of applications between
platforms. The software architecture for a typical JME application is shown in Figure 1.

Figure 1. Java Platform, Micro Edition (Java ME)

The Real-Time Specification for Java (RTSJ)

The dynamic nature of the Java runtime environment is one of its greatest strengths in the traditional desktop and server world. However, the nondeterminism introduced by garbage collection, runtime class resolution, etc., is a real problem for real-time developers. Thus, the Real-Time for Java Expert Group (of which aJile is a member) was formed in March 1999 under the Java Community Process to create a Real-Time Specification for Java (RTSJ) [2]. The RTSJ provides enhancements to the Java Language Specification [4] and Java Virtual Machine Specification [7] in seven key areas: Thread Scheduling and Dispatching; Memory Management; Synchronization and Resource Sharing; Asynchronous Event Handling; Asynchronous Transfer of Control; Thread Termination; and Physical Memory Access. As an example of these enhancements, the RTSJ defines new types of memory areas separate from the Java heap, including ImmortalMemory and ScopedMemory. Real-time threads can create Java objects in these alternative memory areas in the normal fashion (i.e., through the new operator), but since
these objects do not reside in the Java heap, object access does not suffer from nondeterministic garbage collection pauses.

The aJile architecture for efficient real-time embedded Java systems
aJile Systems was founded to meet the need for low power real-time embedded object-oriented application deployment. The aJile Java architecture was designed with an embedded systems focus, and thus maximizes the amount of runtime data that can be ROM'ed, and eliminates runtime instruction stream modification ("quickizing"). aJile CPU's directly supports the Java thread model in hardware, yielding extremely fast thread switching times. The aJile architecture also defines a set of "extended" instructions for physical hardware interfacing and other systems programming tasks; these extended instructions are not available to untrusted dynamically downloaded code.

The JEMCORE-III direct execution Java microprocessor core
The aJile Systems JEMCore-III is a third-generation low-power direct execution Java microprocessor core. The JEMCore-III implements the entire JVM bytecode instructions in silicon; the only two bytecode instructions that trap immediately to software are multianewarray and athrow. Obviously, operations like class loading are handled in software, but once resolution has occurred, execution of bytecodes like invokevirtual are done as single JEMCore instructions, including lock detection. The Java bytecode direct execution improves Java execution efficiency by eliminating the Java interpreter layer and the RTOS kernel layer. Since JVM bytecode are executed as native instructions, the JEMCore’s Java performance is similar to RISC processors executing compiled C. In addition, Java threading primitives (wait, yield, notify, monitor enter/exit) are implemented as extended bytecodes, eliminating the need for a traditional RTOS. The result is extremely low executive overhead with thread to thread context switch times of less than 1µsec. For Java based multimedia-rich, servo motor, robotic applications, it has enhanced with a dedicated microcode-based DSP data-path to accelerate various algorithms for audio, embedded control, servo motor control, voice and hand writing recognition. The figure below illustrates the simplified block diagram of the JEMCore-III.
Hardware support for real-time Java threads

One of the unique features of the aJile architecture is its hardware support for real-time Java threads. Concurrency control is deeply ingrained in the Java Virtual Machine specification; elementary operations such as the method invocation instructions require the acquisition of a lock if the target method is declared as synchronized. Thus, aJile CPU's implement the basic synchronization and thread scheduling routines in microcode. This means, for instance, that the yield() primitive in java.lang.Thread is a single extended bytecode. Several benefits accrue from this approach. First, aJile CPU's require no Real-Time Operating System (RTOS) kernel, thus saving kilobytes of memory. Further, multithreading on aJile CPU's is extremely fast. For example, the time required to execute a yield() and resume a different thread is approximately 500 nanoseconds on a 100 MHz aJile CPU. Additionally, aJile hardware supports periodic thread dispatching, and also implements priority inversion control.

A Java runtime written entirely in Java

A unique feature of the aJile runtime environment is that it is programmed completely in the Java language, including substitutes for the many "native methods" in the Java runtime system.
This is made possible by the JEMBuilder application build tool, which provides the substitution of certain method invocations with aJile extended bytecodes. With this approach, there is no need for an assembler, and developers can create Java implementations for the aJile extended bytecode classes (e.g. physical memory access) in a host simulation environment. The result is that the entire aJile Java runtime is written in Java, thus easing both testing and maintenance. Figure 3 provides an example of device level programming in Java. In addition to exhibiting direct access to device registers (via the rawJEM.* methods), this example also shows how device-level code can notify waiting user threads (through the serialThread.pendSerialEvent() calls). Note that there is no need to abandon object-oriented Java programming practice in order to do system-level programming for the aJile environment.
Memory management
The Java Virtual Machine specification includes several bytecodes for memory allocation. However, the Java environment has no free() primitive; rather, memory reclamation is automatic. Accordingly, the aJile environment implements the memory allocation bytecodes as instructions, and implements a simple mark-and-sweep garbage collector in software. The aJile garbage collector is implemented as a Java thread, and utilizes Java synchronization to assure that the heap it is collecting does not get into an inconsistent state. Heaps in the aJile architecture can be allocated per-thread; thus, the garbage collector can be preempted very quickly (< 1 usecs on the aJ-102 operating at 160 MHz) in order to allow a real-time thread (which uses a separate, non-garbage collected heap) to execute.

Interrupt and trap handling
The virtual nature of the JVM specification means that low-level mechanisms such as interrupt and trap handling are left unspecified. So, the aJile architects have provided our own implementation. In keeping with the theme of writing all software in Java, interrupts and traps are handled by static Java methods, which execute on an “executive”, or supervisor, stack. The executive mode of the processor can be thought of as a single highest-priority thread that has its own stack and heap, and whose methods are the various trap and interrupt handlers. As one would expect, maskable interrupt handlers can be preempted by the occurrence of other, higher priority interrupts, if interrupts are enabled in executive mode (by default, interrupts are disabled when the processor enters executive mode). When the higher priority handler is invoked, a new stack frame is pushed onto the executive stack. Exit from that handler returns control to the preempted handler. Executive mode execution completes when the outermost handler returns.

Development Environment Tool Chain
The aJile development environment allows the use of any off-the-shelf compiler that produces Java standard class files. Thus, developers are free to use whatever Java Integrated Development Environment (IDE) they wish. The only custom elements in the aJile tool chain are the JEMBuilder graphical build tool, which produces JEM format binary images from input class files, and low-level debugging tools.
Figure 4. The JEMBuilder application builder

Most Java implementations rely on runtime class resolution, but this is not necessary if all the necessary classes are present at build time. The aJile JEMBuilder application, itself written in Java, computes the class dependencies for a standard format Java class file, and can create target RAM and ROM memory images only for those classes that are needed. JEMBuilder can also eliminate unused methods, fields, and constants, resulting in significantly reduced memory requirements (often by a factor of 10) for dedicated Java applications. Java's safety and security orientation, particularly the avoidance of pointer manipulation, helps make this build-time analysis possible. JEMBuilder also includes a number of additional capabilities, including the substitution of named static methods with extended bytecodes, as well as the mapping of named Java methods to interrupt and trap handlers (as shown in Figure 4). The former capability allows off theshelf compilers to be used; only JEMBuilder need know about the extended bytecodes. Finally, JEMBuilder presents these capabilities in a user-friendly way using Java Swing graphical components.
The most innovative characteristic of the aJile tool chain is that it allows all application and system software to be written in the Java programming language (or other programming languages that compile to standard Java class files). This allows pure object-oriented design techniques to be brought all the way down to the “bare metal”. This “low-level programming in a high-level language” philosophy eases integration, testing, and maintenance, as well as promoting reuse at the system software level. Figure 4 provides a screenshot of JEMBuilder, highlighting the mapping of interrupts to Java handlers.

**Instruction set customization**

aJile CPU’s generally provide a writeable control store, allowing the instruction set to be customized for particular applications. As with other extended bytecodes, these user-defined instructions are introduced into the instruction stream by the JEMBuilder tool via named static method substitution; no compiler changes are required.
Multiple concurrent Java Virtual Machines

The aJile architecture provides hardware support for the concurrent execution of multiple independent Java applications (i.e., multiple main()’s) in a deterministic, time-sliced schedule with full memory protection. Within its bounded execution interval and memory space, each application environment can employ its own threading and memory management policies without threat of intervention by other faulty or malicious applications. Additionally, each JVM has its own executive mode; this allows JVM’s to be assigned their own interrupts, which pend while that JVM is suspended. This "Multiple JVM" (MJM) capability takes the Java "sandbox" security model to the next level, providing hard space and time boundaries for concurrent JVM execution.

```java
int abs(int x) {
    if (a < 0) {return -a;}
    else       {return a;}
}
```

```java
void gps_sig_proc() {
    short i;
    for (i = 1; i< channels; i++) {

        snip..snip..snip

        if (abs(cr[i].igl) > abs(cr[i].qgl)) {
            spl = abs(cr[i].igl) + abs(cr[i].qgl)/2;
        } else {
            spl = abs(cr[i].igl)/2 + abs(cr[i].qgl);
        }

        snip..snip..snip
    }
}
```
**Low-Power Real-Time Network Direct-Execution SOC for the Java ME Platform**

aJile Systems, Inc.  
[http://www.ajile.com](http://www.ajile.com)

---

**Figure 6.** MJM

**aJ-102: A real-time network SOC for Java platforms**

The aJile Systems aJ-102 is the low-power, real-time, networked, direct execution microprocessor for the JME platform. The aJ-102 is designed to power the next wave of the Internet, secured, and smart appliances that require high-performance Java execution, real-time response, DSP, network access capabilities in a secured manner.

The aJ-102 directly executes Java Virtual Machine™ (JVM) bytecodes, real-time Java threading primitives, and a number of extended bytecodes for custom application accelerations. The native JVM bytecode implementation eliminates the typical interpreter or JIT software layers and provides the most optimal Java performance in both memory requirements and execution time. The microprogrammed RTOS and Java threading primitives also ensure fast, atomic executive...
operations like context switching object synchronization, scheduling and interrupt processing and eliminate traditional external RTOS layer.

The aJ-102 features of a direct-execution Java processor core (JEMCore-II™) with an enhanced DSP capability, on-chip 32 KB unified instruction and data cache (I&D), a single-chip 10/100 Ethernet controller, encryption/decryption engine, a single-chip USB OTG controller, an advanced LCD controller, and all I/O peripherals required for many networked real-time embedded devices.

The aJ-102 microprocessor, bundled with Sun’s JME/CLDC or CDC/FP/PBP runtime system, optimizing application builder, debugging tools and evaluation systems provides a complete solution for implementing real-time, networked embedded devices, entirely in pure Java technology. The powerful combination of direct JVM bytecode execution, direct multithreading support, and fully protected multiple JVM environments is ideal for efficient, safe, and robust Java execution and dynamic delivery of the Java based applications / services on-demand over Internet. The aJ-102 is ideally suited to power the next generation of the Internet appliances:

- Networked edge controllers, and smart sensors
- M2M cellular terminals
- Personal navigation devices
- Mobile POS terminals
- Wireless handhelds and webpads

Figure 7. Simplified block diagram of aJ-102
Conclusions
The aJile family of low-power single-chip embedded Java system-on-chip processors provides an efficient platform for embedded and real-time object-oriented software execution. The aJile CPU hardware provides direct support for the entire JVM instruction set and thread model, obviating the need for a Java interpreter or Just-In-Time (JIT) compiler, as well as the traditional Real-Time Operating System (RTOS). aJile’s hardware technology also supports multiple JVM contexts executing on the same CPU, enhancing safety and security by guaranteeing space and time allotments for multiple Java applications.
Combined with a Java Micro Edition (JME) runtime and a back-end target build tool, these technologies allow the development of real-time embedded applications entirely in Java.

Acknowledgments
Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. JEM is a trademark of Rockwell Collins, Inc. aJile and aJ-102 are trademarks of aJile Systems, Inc. All other trademarks are property of their respective holders.

References
Low-Power Real-Time Network Direct-Execution SOC for the Java ME Platform

aJile Systems, Inc.
http://www.ajile.com


