Abstract
This paper presents Embedded warning systems in C language compared with Java. A key aspect of embedded system is the automatic integration of system components. The proposed methodology is oriented towards the application field of embedded warning systems in Java language compared with C language. A wide class of computing systems required such a detailed design of both hardware and software subparts, fall in the category of embedded systems. Java is an object-oriented and platform-independent language. These two main characters make it abstractive to use Java as programming language for the embedded systems to catch up the main two development trends (implementing functionality in software and networked embedded system). Embedded system is computer-based system, which controls a running operation and process data fast enough to deliver vital results in time. Almost all devices are controlled by embedded systems. Many of these systems have to fulfil quite strict in terms performance and cost efficiency. Java is a promising programming language for the embedded systems. But current it's not ready for the embedded system. There is a lot of confusion these days over whether Java will be a compiled or interpretative language or both. C language is written for embedded systems. Advanced C/C++ optimizations dramatically reduce application memory usage and increase the flexibility and performance.

Keywords
Real-time, systems, embedded systems, object-oriented, platform-independent language, Micro C/OS-II, acoustic calculations, suppression noise-system.

1. Introduction
All embedded systems contain some type of inputs and outputs. The inputs to the embedded system usually take the form of sensors, probes communication signals or control knobs and buttons. The outputs are typically displays, communication signals or changes to the physical world. In the last years indicate a great increase in using embedded systems. Embedded real-time systems are now omnipresent: from automobiles, aircrafts, elevators, buildings to laser jets, microwaves, cellular phones and TV-sets. Embedded systems are becoming more and more complex. One of the ways to manage it is to distribute the system functionality across several low cost microprocessors, which communicate via a shared medium. Communication is essential to achieving a dependable distributed embedded system so designers of these systems are faced with selection of properly working communication. It is almost always the case that the outputs of the embedded systems are a function of its inputs and several other factors such as elapsed time, current temperature (see figure 1). It seems that the number of embedded systems will continue to increase rapidly. Already there are promising new embedded devices that have enormous market [1]. These system models need specification for their time requirements, event handling, communication, concurrency and synchronisation. Because real-time systems are often distributed, the models also have to show the distribution of the system. A real-time system is highly integrated with its physical environment, in that it receives events and information through sensors and controls devices through actuators. Thus, the real-time system works closely with specialised hardware and has to handle low-level interrupts and hardware interfaces [1].
Interfaces between software and hardware and between hardware blocks can be easily described either at the transaction-accurate level or at the cycle-accurate level. Moreover, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. A system that involves closely integrated hardware and software is called an embedded system. Embedded system can be found in cars, electronics units and many other devices almost every where. An embedded real time system often uses a small real-time operating system that can be fitted easily into a system with limited memory capacity and are computer-based systems, which controls a running operation and process data fast enough to deliver vital results in time.

Embedded system always involves both the software and hardware co-development. Clearly, the complexity of real time systems and the need to design precise and descriptive models is more important than normal systems [1].

<table>
<thead>
<tr>
<th>Factor</th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Time</td>
<td>Relative large</td>
<td>large</td>
</tr>
<tr>
<td>Cost</td>
<td>Application Dependent</td>
<td>Large</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>Low</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 1: Hardware vs. Software Implementation

There are implementation issues (see table 1) such as that the environment must have adequate support for real-time constructs (process or threads, communication and synchronisation mechanism). There are also times when the implementation of a model has to be optimized to allow for maximum performance. The implementation of parallelism is a big topic in embedded computer system both hardware and software that is basically independent of concurrent programming. [1] Concurrency execution threads of control different parts of the software run in parallel, the concurrency can be performed in real parallel system with several processors, each thread executes on its own processor. Virtually embedded systems are inherently concurrent. Concurrent programming is programming of notations and techniques for expressing potential parallelism and solving the resulting synchronisation and communication problems in embedded systems. High requirements on most of the non-functional requirements such as reliability fault tolerance and performance. Non deterministic, it is impossible to formally prove that the system will work in all situations under all conditions due to complexity of concurrency.

external asynchronous events that may occur in any order where the hardware involved. Real-time systems are often divided into hard and soft categories. In hard real-time system, the response is considered an unacceptable error that can result in loss of life. Examples of hard real-time systems are the airplane control software, An automatic train control and other objects where something very bad that will happen if the computer real-time system dose not deliver it’s output in time. Soft real-time systems can accept a late response time; for instance, in digital telephone systems, it may takes long time to connect a call or the connection fail, the result is not considered a serious or dangerous error. The key in an embedded system is the time constrain, the simplest time constrain that an embedded system can have is the requirement to act and recognise. In general, time is a restriction on the time process to wait for communication. If an embedded system is designed well, the existence of the processor and software could be completely unnoticed by user of the devices, which based on real-time systems [6]. Hardware/Software Co-Verification is typically performed at a low level of abstraction, using an Instruction Set Simulation (ISS) model of a CPU in conjunction with a Verilog, VHDL and micro C/OS-II model of the rest of the design. The CPU subsystem will be replaced by an RTOS simulator and application code written to the Application Programmers Interface (API) of the RTOS. Verilog or, VHDL or micro C/OS-II is still used to model the rest of the design [9]. Embedded systems rely on a close, integrated relationship between software and hardware. However, in a traditional embedded systems design cycle, hardware and software are often developed separately, then integrated after the hardware has been built. Not surprisingly, often the hardware and software do not work correctly together the first time. In many cases the real software and integration debugging begins only after the hardware has been built. This debug and integration can take up to half of the project schedule. The reasons to use embedded systems are many but the most important are flexibility and performance (see figure 2).

Figure 2: Flexibility vs. performance
2. Outlines
This paper provides the following sections. In section 3 provides embedded system and main elements. Section 4 provides embedded system in Java. Section 5 provides programming in Java Stamp. Section 6 provides implementation of a noise suppression system. Section 7 provides embedded system in C. Section 8 provides the results. Section 9 provides dissections. Section 10 provides conclusion. Section 11 provides future work. Section 12 provides my all acknowledgements. Section 13 provides the references.

3. Embedded system and main elements
The main elements in embedded systems are the following:

3.1 Communication
When passive objects communicate with each other, normally, is defined through operations by synchronous message sending. [6] One object calls in an operation another operation by sending message between them, the calling object waits for the operation to execute, later to have control return when operation is done. As part of the operation call, information can be passed between the object as parameters. Number of mechanism can be used to active object to communicate with each other [6]. Some of most well common mechanisms are used when communicating between active objects are;
- Operation calls (synchronous messages).
- Mailboxes, a technique that defines mailboxes or messages.
- Shared memory, a block of memory is reserved for communication, so that two or more active object can write and read information. The block has to be guarded by some kind of synchronzation mechanism. So that the concurrently executing objects do not try to write and/or read the information at the same time.
- Remote producer calls (RPCs), the RPCs handle the distribution of concurrent threads to separate computers in a network and allow the threads to be write in different languages.
- Rendezvous, a technique by which specific rendezvous points in the execution of two threads are defined, the first thread to reach its rendezvous point stops and waits for the other thread to reach its corresponding rendezvous point and synchronous. Asynchronous communication is unpredictable Communication can be asynchronous; that is, an event can occur at any time in the execution of the system and it is not possible to know when a specific event will be sent. [6] Synchronous communication is predictable in that it can occur only at specified times in the execution of the system. The real-time system uses both a synchronous and synchronous communication. The structural description features modules and ports are also useful for system design, as are the extensive set of data types and the ability to express concurrency using processes. [6] The mechanism for communication and synchronisation the hardware signal is not sufficiently general for system-level modelling. For example, in a system level design, a designer might want to specify that several modules communicate using queues, or that several processes execute concurrently and manage access to shared global data using mutexes. [6] A new set of features for generalised modelling of communication and synchronisation. These are channels, interfaces and events. A channel is an object that serves as a container for communication and synchronisation. Channels implement one or more interfaces. An interface specifies a set of access methods to be implemented within a channel, but the interface itself does not provide the implementation. An event is a flexible, low-level synchronization primitive that is used to construct other forms of synchronization. Channels, interfaces and events enable designers to model the wide range of communication and synchronisation. “Petri-nets” are an established operational modelling tool used to model the control of real-time, distributed, or parallel systems. Petri nets are similar to block diagrams and composed of input places, output places, transitions, and tokens [6]. Tokens are used to simulate the dynamic activities of a system, input

![Figure 3: Embedded system Concept](image-url)
must be taken to safely interact with other environments. For example, an interrupt handler must never block and a thread must disable interrupts before interacting with a data structure that the thread shares with an interrupt handler. A simple communication example is a FIFO, which stores ten characters. [6] The FIFO will have blocking read and write interfaces such that characters are always reliably delivered. First, specify the separate read and write interfaces to the FIFO [6].

3.2 Synchronization

Synchronization is the process of co-ordinating concurrently executing threads with each other, so that they do not improperly interfere with each other (e.g., that they are not concurrently trying to modify or access a shared resource at the same time) and that they interact in an efficient order. Synchronization is necessary in any system that has concurrency [5,6]. Not only are the active objects concerned with this, but also all passive objects that are shared between active objects. The problems that can occur if synchronization is not properly handled are:

- Incorrect shared access: Resources or passive objects that are shared between active objects can cause problems if they are accessed concurrently. If one thread calls an operation in a passive object, the operation can execute some code lines that set attributes in the object. [6] Anytime during the execution of the operation, the thread can be suspended by the operating system, which then schedules another thread for execution. [6] This other thread calls the same or another operation in the same object, now on an object that is not in a safe state (since the last call to that operation hasn’t finished yet). Let’s say that the operation executes to the end and that the first thread then is scheduled, resuming execution at the point where it was suspended. The attributes of the object have changed, and the final result of the operation is undetermined. [6] What is needed is mutual exclusion between the threads, so that only one thread at a time executes the operation and no other thread can access that operation until the first thread has finished it. [6] The problem with shared access concerns both the objects that are shared as well as other shared resources (devices such as printers, communication ports, etc.).

- Inefficient resource usage: Often there are dependencies between the executions of different threads. One thread might depend on another thread to make some preparation of the data it is using. [6] The first thread could be continuously executing in a busy-wait loop, checking over and over again to determine whether the other thread is finished. This is a big waste of machine resources. A better solution is to have a synchronization mechanism to co-ordinate the thread with the preparing thread, to ensure it is not scheduled to execute until it has been notified that the first thread has finished its work.

- Deadlocks: A deadlock occurs when a number of threads all are waiting for each other. As an example, a system could have two communication ports, A and B, guarded by some synchronization mechanism. [6] Two threads (possibly two instances of the same active class) at some point need both of these communication ports. One thread manages to reserve port A, and the other manages to reserve port B. The first thread now waits for port B to be released, and the second thread waits for port A, while neither of them is releasing the port it has reserved. This is a deadlock, which totally stops the system and stops continuing to run correctly.

- Starvation: A starvation problem is when one thread never gets to run. It occurs when the priorities of the threads are defined in such a way that it is impossible or very difficult for one thread to ever get control. Consequently, the task of the thread is never performed, which in turn leads to other difficulties. [7] The mechanisms used to synchronize threads are concepts such as semaphores, monitors, or critical regions. They all share the common characteristics that they guard a specific code block so that only one thread at a time gets access to it. The code block then contains the code to reserve or use a resource such as a device or a shared object. The mechanisms also have operations to wait for a resource (or actually, the code block guarding the resource) to be released without continuous “busy waiting,” which is waste of processor resources. The mechanisms need to be supported by the operating system because they have to be implemented using in-divisible low-level instructions, and several programs executing concurrently are involved. Resolving the synchronization problems described also calls for the scheduling of the threads, which is done through setting priorities. [7] Obviously, a thread with higher priority has precedence and is scheduled to run before a thread of lower priority.

In some operating systems, it is also possible to parameterize the scheduling algorithm or even to choose the scheduling algorithm. The simplest way to control the scheduling is in the code of the active objects, where they can give up the control by calling functions. Another example a sleep (which makes the thread idle for a specified amount of time) or by waiting for a communication or synchronization mechanism. For example a mailbox or semaphore to be signalled or released. The built-in support for synchronization in UML is limited, but it can be dined through extensions using stereotypes or properties. [7] A class or an operation can have its concurrency requirements
defined through a Concurrency property with the possible values Sequential, Guarded, and Synchronous. If a property is set for a class, it effects all operations in the class, otherwise only the operation for which the property (is set) is effected. The meanings of the values are:

- Sequential: The class/operation is intended only for use in a single thread of control (not concurrently by several threads).
- Guarded: The class/operation will work in the presence of multiple threads of control, but it needs an active collaboration of the threads to achieve mutual exclusion. The threads normally have to lock the object/operation before using it, and unlock it afterward.
- Synchronized: The class/operation will work in the presence of multiple threads of control; the class itself handles this. This feature is being built in more modern programming languages, such as Java, where an operation can be declared as synchronized; then the handling of synchronization of that operation is automatic so that only one thread at a time can access it. If synchronization must be modified more explicitly, a semaphore class can be defined and instantiated to semaphores whenever the synchronization between active objects has to be shown. Another possibility is to define a stereotype semaphore that is used for all classes whose instances are shared and should be guarded by a semaphore (in that case, the semaphore is implicitly connected to the class that is stereotyped). Synchronization mechanisms other than semaphores can of course be modelled as classes or stereotypes as well.

3.3 Scheduling
A technique which used to synchronise active objects is to schedule them well. Scheduling is the process of deciding which thread should run next in a situation where several threads are conceivable. [2,5] Scheduling is done by an operating system, which allocates the processor to thread using some a round–robin algorithm, by every thread gets a specific time slot or dynamic priority algorithm which is based on giving priority to the threads that are in the interacting with user. The programmer can control parts of the scheduling, normally by setting priorities of the threads or by setting parameters of the scheduling algorithm. [2] In a complex situation, detailed control is needed of the scheduling to handle (supervisor thread) the thread to be designed. Supervisor thread (handler thread) is given maximum priority that control to the application in the supervisor code [5]. Normally, the priority set during modelling needs to be tuned when testing prototypes or early release of the system. Most real time systems are priority scheduled based. The priority for each task is application specific. The control of CPU is always given the highest priority task ready to run.

3.4 Modules
This is a hierarchical entity that can have other modules (a notion) or processes contained in it. [4,5] Modules and processes can have a functional interface, which allows hiding implementations details and, for this, including blocks of IP.

3.5 Signals
Signals are passed between objects in the system, either synchronously or asynchronously, and are messages (only the message passed between communicating objects also contains a signal object, and is not only an operation call). Signals are described in class diagrams, and all the signals in a real-time system are modelled in a hierarchy. This makes it possible to group a number of signals by having a common super class. A receiver of a signal can then specify that it accepts objects of a specified super class, which then means that it accepts signal objects of the super class or any of its specialized subclasses. A super class in such a hierarchy can also contain general information about the signal, such as a priority, time sent, and so on. As signals are a special case of events, they can be divided into logical or physical signals carrying either low-level hardware information or higher-level interpretation of what the event means in terms of the execution of the system.

3.6 Events
In a real-time system, events are drivers of system activity. An event is something that occurs in the system or in the environment, and something to which the system must react to and handle. All events that can occur must be defined; further more, the system’s behaviour when the event happens must be defined. The behaviour is often connected to the state of an object in that the same event can cause different behaviours depending on the state of the object when the event occurs. An event can also cause a transition from one state to another. In UML, there are four different categories of events:

- A condition becoming true: A guard-condition that becomes true is considered an event. This is typically implemented through an active object that monitors this condition and notes when it has become true. [5]
- The receipt of an explicit signal object: A signal object is sent from one object to another. The signal object, which is considered a message, is of a signal class and may contain attributes and operations. Signal objects are typically sent through some operating system mechanisms, such as a mailbox or queue. Signal objects can be sent asynchronously.
- The receipt of an operation call: One object calls an operation in another object. [5] This operation call, which is considered a synchronous message, passes information through parameters and the return is a value.
A passage of time: A specified amount of time passing is also an event, some-times called a periodic event. Such an event is used to describe an intended delay in processing or time-outs if some other event hasn’t occurred within a specified time. A passage-of-time event is implemented either through a sleep call, which suspends the execution of the thread for a specified amount of time, or a request to the operating system of a time-out message, which is then sent to the requesting object when the specified time has elapsed. Events can be divided into logical and physical categories. [5] A physical event is low-level, at the hardware level; a logical event is a higher-level representation of that occurrence. The physical events for example “interrupt” could be translated into the logical event “alarm from the infrared sensor device.” Since logical events are at a higher level of abstraction, the models preferably should be defined with logical events rather than low-level physical events. Physical events should be translated into logical events as close to their detection as possible (through an active object with a state diagram).

3.7 Messages

Objects communicate through messages; specifically, messages are used to communicate between passive objects, between active and passive objects, or between active objects. [5] A message is implemented by a simple operation call, or as a signal object that is put in a mailbox or queue. [1] They are defined as follows:

• Simple: Represents a flat-flow of control. A simple message shows how the control is passed from one object to another without describing any details about the communication. This message type is used when details about the communication are not known or not considered relevant in the diagram. It is also used to show the return of a synchronous message; it is drawn from the object that handles the message back to the caller to show that the control is passed back (possibly along with some result).

• Synchronous: A nested flow of control, typically implemented as an operation call. The operation that handles the message is completed (including any further messages being sent as part of the handling) before the caller resumes execution. [2] The return (can be shown in a sequence diagram) as a simple message, or the return can be implicit when the message has been handled. Synchronous message between active objects indicates wait semantics; the sender waits for the message to be handled before it continues to execute. Simple and synchronous messages can be combined into one line, with the synchronous message arrow at one end and the simple return arrow at the other. This indicates that the return of control is virtually immediate. [1] Other message types are extensions to the basic UML, such as a balking message (a message that is sent only if the receiver is ready to accept it) and a time-out message (that will be cancelled if it is not handled within a specified amount of time). These message types and other similar variants are described as stereotypes of messages [4].

3.8 The processors role in embedded system

Processors are the most important component in an embedded system. Embedded processor selection is important step in embedded design such in a system-on-chip (SoC), processors can be integrated along with other hardware on the chip. Two main different processors architecture are the Von-Neumann architecture, which is sequential and the Harvard architecture which is Parallel (see figure 4 and 5) [14]. The programmers should have a good view of the processor and hardware interfacing view. Processes are also used to describe functionality and can be stand-alone entities or can be contained inside modules [4,5].

![Figure 4: Von-Neumann Architecture (Sequential)](image)

![Figure 5: Harvard Architecture (Parallel)](image)
Three different process abstractions to be used by hardware and software designers: asynchronous blocks, synchronous and asynchronous processes. A discrete processor considered as an independent component while an integrated processor considered as "processor plus other hardware implemented in chip".

4. Embedded system

The main embedded systems language in this paper are Java/Javelin/Java stamps and C language.

4.1 Embedded system in Java

The Java programming language is robust and versatile, enabling developers to: Write software on one platform and run it on another, to Create programs to run within a web browser, develop server-side applications for online forums, stores, poll and Write applications for cell phones, two-way pagers, and other consumer devices. However, there are techniques that can be use to boost Java. To make better-informed decisions about using Java by understanding the factors that affect its performance and selecting meaningful benchmarks for embedded applications. Techniques for improving application execution and choosing the right Java virtual machine address only a few aspects of system architecture that affect overall Java performance[11]. When selecting an embedded Java platform, must take into account a host of other factors that have an impact on performance. Among them are hardware processor selection, Java compatibility and supported APIs, application reliability and scalability, the choice of a real-time operating system (RTOS) with associated native libraries and drivers, the availability of Java development tool kits and middleware, graphics support, and the ability to put the application code into ROM. Once hardware and software development platform selected, there are a variety of factors to consider that will help you choose the best-performing Java virtual machine for your application. Although the average Java bytecode application executes about ten times more slowly than the same program written in C or C++, how well an application is written in Java can have a tremendous impact on performance (see table 2).

Applying statistical analysis to the performance data for the programs revealed that actual performance differences depended more on the way the programs were written than on the language used. Indeed, the study showed that a well-written Java program could equal or exceed the efficiency of an average-quality C/C++ program. Various approaches are available for boosting bytecode execution speed. They include using a just-in-time (JIT) compiler, an ahead-of-time compiler, or a dynamic adaptive compiler; putting the Java application code into ROM; rewriting the JVM's bytecode interpretation loop in assembly language; and using a Java hardware accelerator. JIT compilers, which compile bytecode on the fly during execution, generally aren’t suitable for embedded applications, though. They produce excellent performance improvements in desktop Java applications but typically require 16 to 32 MB of RAM in addition to the application’s requirements. The large memory requirement places JIT compilers out of reach for many categories of embedded applications. Ahead-of-time compilers rival JIT compilers in increasing Java execution speed. Unlike JIT compilers, they’re used before the application is loaded onto the target device, as their name indicates. That eliminates the need for extra RAM, but it creates the need for more ROM or flash memory (storage static and memory), because compiled machine code requires four to five times the memory of Java bytecode. Compiling ahead of time tends to undermine one of the great benefits of the Java platform because a measure of dynamic extensibility can be lost, since it may not be possible to download new versions of compiled classes. Additionally, any dynamically loaded code, like an applet, won’t benefit from ahead-of-time compilation and will execute more slowly than resident compiled code. Profiling Java code, although somewhat complex, can help minimize code expansion when using an ahead-of-time compiler.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Medium requirements</th>
<th>High-end requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 to 150 Pyrhoezene MIPS &lt; 100 MHZ clock frequency &gt;9 KB 1D Cache No hardware FPU</td>
<td>150 to 250 Pyrhoezene MIPS &lt; 100-200 MHZ clock frequency &gt;9 KB 1D Cache Hardware FPU optional</td>
<td>250 to 400+ Pyrhoezene MIPS &lt; 200+ MHZ clock frequency &gt;16 KB 1D Cache Hardware FPU</td>
</tr>
<tr>
<td>Memory</td>
<td>Additional RAM requirements</td>
<td>Limited cache of instructions</td>
</tr>
<tr>
<td>2 MB ROM</td>
<td>2 to 4 + MB ROM</td>
<td>&gt; 5 MB ROM</td>
</tr>
<tr>
<td>2 MB RAM</td>
<td>4 to 9 + MB RAM</td>
<td>&gt; 10 MB RAM</td>
</tr>
</tbody>
</table>

| Java Apps | Typically non-graphical | Moderately graphical | Intensively graphical |

Table 2: Typical requirement for Java Systems

A good goal is to compile only that 20 percent of the Java classes in which the application spends 80 percent or more of its time. Dynamic adaptive compilers offer a good compromise between JIT
and ahead-of-time compilers (see table 3). They are similar to JIT compilers in that they translate bytecode into machine code on the fly. Dynamic adaptive compilers, however, perform statistical analysis on the application code to determine where the code merits compiling and where it's better to let the JVM interpret the bytecode [11]. The memory used by this type of compiler is user-configurable, so you can evaluate the trade-off between memory and speed and decide how much memory to allocate to the compiler. Placing the bytecode into ROM can contribute to faster application performance. It doesn't make the code run faster. It does, however, translate the code into a format that the JVM can execute directly from ROM, causing the code to load faster by eliminating class loading and code verification, tasks normally performed by the JVM. Another way to speed up bytecode execution without using ahead-of-time or dynamic compilation techniques is to rewrite the bytecode interpreter in the JVM. Because the interpreter is a large C program to make it run faster by rewriting it in assembly language. Embedded or real-time systems include all those in which timing constrains imposed by the world outside of the computer play a critical role in the design and implementation of the system. Common areas for embedded systems are machine and process control, medical instruments, telephony, and data acquisition. A primary source of input for embedded systems is random, short-lived, external signals. When such signals arrive, the processor must interrupt whatever else it is doing to capture the data or it will be lost. Thus, an embedded program is most often organized as a set of individual, but cooperating threads of execution. Some threads capture new data, some analyze the new data and integrate it with past inputs; some generate the outgoing signals and displays that are the products of the system. Currently, most embedded programs are coded in C, with critical parts possibly in assembler. Some of the major problems of C for embedded systems are: The permissiveness of C operations, which can lead to undisciplined coding practices and ultimately to unstable execution, the absence of universal standards for multithreading, shared data protection, and intra-thread communication and coordination, which can make the program hard to transfer to alternate platforms. The use of Java may be different, however. Java programs for embedded applications will differ from net applets in at least five major ways. Embedded applications will be:

- Compiled into the native ISA for the target hardware.
- Capable of running in the absence of a hard or floppy disk, and a network connection.
- Supported by highly tailored, thus relatively small run-time packages.
- Able to execute on multiple processors, if needed for capacity expansion.
- Contain significant amounts of legacy C code, at least during the transition from C to Java.

Java is an object-oriented and platform-independent language. These two main characters make it abstractive to use Java as programming language for the embedded systems to catch up the main two development trends (implementing functionality in software and networked embedded system) of embedded system. But Java's large code size, slow execution time and undetermination caused by garbage collection (i.e. using software approach or hardware approach to solve garbage collection problem in real-time system) prevented it from using widely in the embedded systems [9,10].

4.2 Java different than other languages

In previous High-level languages, the portable element was the source program. Once the source program is compiled into executable form for a specific instruction set architecture (ISA) and bound to a library of hardware-dependent I/O, timing and related operating system (OS) services, portability is lost[3,11]. The resultant executable form of the program runs only on platforms having that specific ISA and OS. Thus, if a program is to run on several different platforms, it has to be recompiled and relinked for each platform. And if a program is sent to a remote target for execution, the sender must know in advance the exact details of the target to be able to send the correct version. With Java, source statements can be compiled into machine-independent, "virtual instructions" that are interpreted at execution time. Ideally, the same virtual code runs in the same way on any platform for which there is an interpreter and OS that can provide that interpreter with certain multitreading, file, graphical and similar support services. With portability moved to the executable form of the program, the same code can be sent over the net to be run without prior knowledge of the hardware characteristics of the target. Executable programs in the Java world are universal. In principle, portability could have been achieved in the C or C++ world by sending the source program over the net and then having the compilation and linkage done as a pre-step to execution. However, this approach would require that the target system have sufficient CPU speed and disk capacity to run the sophisticated compilers and linkers required. Java is not just a new concept in portability. The Java language evolved from C and C++ by locating and eliminating many of the major sources of program error and instability, i.e. C has an element know as
a pointer that is supposed to contain the address at which a specific type of information is stored. Java does not have pointers. Equally important, Java has built-in support for multithreading. C and C++, were designed to express a single thread of computing activity. There are in fact, several proposed standards, so there is no standard. Java bypasses the problem by building multithreading and the data synchronization it requires directly into the source program. The Java embedded Communication such as mealy machine; the output is based on the input and the present state. There are three models [3, 9]:

- A time independent model (Mi).
- A time-relative model (Mr).
- A time-absolute (Ma).

In the time independent model (Mi), there is no notion of time, there can be no assertion of the length of time required for input and output signals. For example, if inside the Mealy machine is a counter, an input event sets off the counter, then another input event comes in, there can be no knowledge as to how many counts it has accomplished. In this model there is no order dependence. This non-deterministic behaviour can be eliminated with a handshaking protocol. The time-independent model is a good design choice when there are critical data transfers or when the targeted implementation platform is unpredictable (deep-submicron interconnects). The second model variant requires a relative notion of time, which means that there is some idea of at what point one event occurs with respect to another. This type of model can be used when the overhead of handshaking is too expensive or not necessary. The partial ordering of events might be described by an approximate number, a min-max range, or a distribution. The third model is the absolute time model, which is mapped directly onto a timeline. This third model can be seen as the time-relative model with the defined origin and value described in seconds for the event orderings. Refer to the Prechelt study which determined that the average memory requirement of a program written in Java is two to three times greater than for one written in C/C++ [10,11]. Even the compact nature of bytecode, usually about 50 percent smaller than compiled C/C++ machine code, can’t offset that overhead. Recognizing that trying to drop Java in its original, desktop-oriented form into embedded systems won’t work, Sun Microsystems, Java’s originator, took the language through several evolutionary steps in an effort to tailor it to the embedded environment. Java components that affect RAM requirements include the JVM (for bytecode execution), the potential dynamic compiler, the Java heap, and the number of threads (the latter two obviously depend on the application). Executing as much of the application as possible using an interpreter while maintaining acceptable execution performance helps contain the memory footprint. Selecting a highly scalable operating system and C run-time package allows tuning these software components for optimal memory efficiency. Scaling the Java environment can be complex, however. Usually, a two-stage process is involved [18]. First, using the command line verbose option, java -v, to see the classes an application uses and then manually extract the needed libraries and classes. If this process doesn’t save sufficient space, then use filtering tools, like Java Filter from Sun’s Embedded Java platform. By using Java, there is an expectation to increase memory and CPU resources compared with using C/C++. Many programming languages provide garbage collection to reduce the need for memory management related programming. Traditional garbage collection techniques do, however, lead to long and unpredictable delays and can therefore not be used in real-time programs. In a real-time program, the maximum time for elementary operations (allocating an object and accessing an object) must be small. In order to guarantee this, garbage collection must be performed in parallel with the program and must be scheduled in such a way that the free memory never becomes exhausted.

4.3 Special hardware for embedded applications

Java programs currently execute on PCs or workstations, which of course have a hard disk. Thus, the Java interpreter expects to be able to load any needed class libraries and code components from a disk. In contrast, many embedded systems are housed within a highly-constrained box, and have to operate in the harsh factory floor environment. Lack of space, power or cooling, or excessive vibration may preclude a standard hard or floppy disk As a result, we anticipate that embedded applications will be structured as a small section of initialization code that starts the Java programs from a ROM disk. (A ROM disk behaves like a hard disk, but uses read-only-memory as the medium.) any needed temporary data files will be stored in RAM disks [13]. Java applets may be called upon to give voice to sound files and sight to arbitrary still or moving picture files. Thus, the Java runtime operating system must provide hardware support for a wide variety of potential audio-visual operations. The Java applet environment places open-ended requirements on its OS. Embedded applications are generally not so free wheeling; by the time the application is ready for the field, the OS requirements and hardware environment should be known. Thus, the OS needed for an embedded Java applications can be tailored to that specific
application, by including only what is known to be needed. The saving in memory size and maintenance effort can be enormous.

5. Programming-in-Java-Stamp

Parallax provides a special board that contains a small breadboard, the Javelin Stamp is probably not fast enough to record the sound directly, that is a big problem in work with acoustic electronic embedded system [15,16]. Programming the J-Stamp is like using any Java IDE. The provided software allows creating Java source files and set options for the compile process. The reference to System.out.println would be a problem since J-Stamp has no place to print a message. However, assuming the serial cable is in place, the IDE will display the message in a special window. The problem with the original code is that it specifies the standard main function that takes a String array as an argument. With JavaScript there is no way to provide a command line argument, so it looks for a main function with no arguments. Java, like C++, allows creating different functions that have the same name, differing only in their argument lists. To uniquely identify a function like main, you have to include its argument list. A main() function is not the same as main(String [ ]), and since J-Stamp's runtime code looks for main 0, you must supply that function. Not much of a change, but still a difference between the two versions. Of course, if you really didn't want to change the existing code, you could write a stub main method that calls the main(String []) method. One nice feature of J-Stamp is the debugger. While it isn't as powerful as a Java debugger on a workstation, being able to track the flow of your program really improves your productivity. Java Stamp, however, has far more differences from standard Java. In particular:

- The int type is 16 bits wide, instead of 32-bits.
- The long type is not supported.

Java/J-Stamp does not support unsigned math. This causes problems with constants that have their high bit set (i.e. 0xFA00) because the compiler will recognize this as too large to fit in a signed integer. With the 8-bit byte data type, the values range from -128 to 127. If you need unsigned bytes, use char which can range from 0 to 255.

- Floating-point types (float and double) are not supported.
- There is no garbage collection. Once you allocate memory, it is never reclaimed.
- Many standard Java class libraries are not available; others are different (because of data type differences, for example). In addition, JStamp has many libraries that don't appear in standard Java. These allow you to control the hardware and peripheral devices.

- J-Stamp only supports one thread. However, you can manage multiple tasks with the Timer object.
- J-Stamp strings and characters are composed of 1-byte ASCII characters, not 2-byte Unicode characters.
- Interfaces are not available, although you can create abstract classes that other classes can extend.
- The J-Stamp only supports single dimensioned arrays.

However, for embedded programming systems the performance hit of floating-point math, and the overhead of Unicode are the main points. The real difference, of course, is what each device adds to standard Java. J-Stamp has a variety of classes to control digital I/O and other specific microcontroller tasks. One of the most interesting things about the J-Stamp is its use of Virtual Peripherals (VP). A VP is a built-in object that runs on a periodic interrupt. JStamp reserves a fixed number of slots for VPs; it is allowed only a limited number active at once. VPs allowing to do background processing without having to use threads (which J-Stamp does not support). Perhaps the most common VP is the UART. This allows turning any one of JStamps 16 I/O pins into a serial transmitter or receiver. For example to transmit and receive, should have to use two VPs. Since the VPs operate as interrupt routines, the UART class can buffer characters.

5.1 Java-Stamp

Since J-Stamp does not use 1-Wire, an example of the code program. The main code would be as follows:

```java
while(true){
    if(CPU.readPin(CPU.pin0))
        CPU.writePin(CPU.pin3,true);
    if(CPU.readPin(CPU.pin1))
        CPU.writePin(CPU.pin3,false);
}
```

This sends the current state to another device using the serial port. Creating the Uart object automatically creates a VP. Java Stamp can use its other sophisticated I/O commands to read a variety of sensors. For example, the CPU.pulseIn can measure a pulse width (useful for reading a tachometer, for example). You could use CPU.rcTime to measure the time it takes for a capacitor to charge (or discharge) through a resistor. It should be extremely interesting to watch
the embedded Java space over the next generation and to see how an increasing number of users will take advantage of its many great features. A common current use for Java is as a user interface engine; the real-time and device-specific stuff is still being done in C. A Java Virtual Machine is simply added to the system for executing complex and, perhaps, field-upgradeable, user interfaces threads. The big advantages are that: (1) any competent Java programmer can be enlisted to produce that part of the embedded software, and (2) feedback from marketing and target customers can be sought and incorporated long before the actual hardware is complete. Another option is to base system design on a Java processor and write all of the firmware in Java. The Javelin Stamp comes with its own IDE, which has debug features that support multiple breakpoints. The Javelin has 32K for program/data/storage space. Chances are not be able to use Java code designed for other systems, like a PC etc, without some modification. The Javelin does not support garbage collection to embedded systems and real-time processing [10,11]. Many existing Java classes for the PC, etc rely heavily on garbage collection. Working with the Javelin without garbage collection (this is not a detrimental factor) the code will become more efficient and execute faster. The code that stored in EEPROM will transfer into RAM chip, which is on the Javelin Stamp before the program executes. This allows; faster execution speeds and the EEPROM allow retaining the program while the power is off.

5.2 J-Stamp board description

The Javelin Stamp is a 24-pin DIP module programmed in a subset of the Sun Microsystems Java language. Java is a trademark of Sun Microsystems, Inc. The Java programming language is not the only thing that makes the Javelin Stamp uniquely different from BASIC Stamps: 32 k bytes of RAM/program space (leftover space can be used for variables, arrays, and serial buffers; 32 k bytes of non-volatile EEPROM; buffered serial communication in the background; up to seven UART objects that can communicated independently of each other and the main program; pulse width modulation in the background; 8000 instructions/second execution speed (not including background processes which run independently of foreground tasks). The Javelin Stamp is compatible with any Basic Stamp programming board with a 24 pin DIP socket. The Javelin Stamp works with all of the power supplies that carried in stock [15,16].

Figure 6 indicates the embedded systems data flows from sensors to the central cluster of processors, between processors in the central cluster and from processors to actuators and output displays as alarm, communication overheads adds to the computer response time. The use of CPU, sensors (microphones) and other instrumentation for embedded warning system offers the possibility of reducing complexity. It is possible to connect all the sensors to the CPU with a single wire (with ground return, of course), this single wire called a data bus will increase the complexity. To understand the operation of data bus, refer to the system block diagram. The CPU controls the use of the data bus (wires) by signalling each sensor through transmitter/receiver T/R units. A T/R unit responds only to one particular sequence of pulses, which can be thought of as the address for that unit. When a T/R unit receives data corresponding to its address, it activates an analog-to-digital converter. The sensors (microphones) analog output at this instant is converted to digital binary numbers. These numbers and the T/R unit’s address are included so that the CPU can identify the source of the data. Thus, the CPU interrogates a particular sensor and then receives the measurement data from the sensor on that data bus. The CPU then sends out the address of the next unit whose sensor is to be sampled. A comparator circuit used to check which value is larger. By design the circuit so that he comparator output goes to 5 V if one value is larger and 0 V if the other value is larger then substitute the comparator circuit’s output with that of the switch (see figure 7).
import stamp.core.*;
public class ButtonLED {
    static boolean PO = true;
    public static void main() {
        while(true) {
            if (CPU.readPin(CPU.pins[1]) == false)
                PO = !PO;  // Negate PO
            PU.writePin(CPU.pins[0], PO); // LED [On]
            CPU.delay (1000);
            }  // end if
            else {
                CPU.writePin (CPU.pins [0], true); // LED [Off]
            }  
            }
        }
    }

The Javelin Stamp comes with its own IDE, which has debug features that support multiple breakpoints. The Javelin has 32K for program/data/storage space. Chances are not be able to use Java code designed for other systems, like a PC etc, without some modification. The Javelin does not support garbage collection to embedded systems and real-time processing. Many existing Java classes for the PC, etc rely heavily on garbage collection. Working with the Javelin without garbage collection (not a detrimental factor) the code will become more efficient and execute faster. The code that stored in EEPROM will transfer into RAM chip, which is on the Javelin Stamp before the program executes. This allows; faster execution speeds and the EEPROM allow retaining the program while the power is off. In general, embedded systems are hard to program, and expensive to maintain. Development takes a long time. The C language has been around a long time. The language Java, appropriately applied, has been well-documented to reduce development time and expense and ease maintenance, the language of the new millennium Java. It is the physically smallest, lowest cost, lowest power embedded Java system we could think of that was still very powerful and useful. JStamp will support 1Wire through a serial port and a DS2480B. It will be switchable in firmware between 1Wire and RS232. This will be implemented on a JSimm card called JSimml-Wire. The core Java and the Java Virtual Machine (JVM) are based on desktop computers and the Internet. Soon, Java will be available for programming Internet devices, such as the Web-TV, Web-Radio, NC, NetPC, etc. known as Embedded Java. Still, Java is far from real-time and lethal for small embedded devices. However, the Java language is almost ready, but the runtime environment and classes are killers. Fortunately, these will become faster in the future. Here are some key advantages that Java provides. 

**Portability**: Platform independence enables code reuse across processors and product lines, allowing device manufacturers to deploy the same applications to a range of target devices. This can provide manufacturers with a cost-effective, **Rapid application development**: The Java programming language offers more flexibility during the development cycle. Development can begin on a variety of available desktop environments, well before the targeted deployment hardware is available. Many Java technology-based desktop development tools can use components created for a variety of applications, **Connectivity**: the Java programming language provides a built-in framework for secure networking. Some software vendors offer off-the-shelf applications, such as browsers and email clients for Internet-connected devices, **simplified support and maintenance**: with embedded Java technology, applications are easy to maintain, resulting in greater longevity. There is excellent availability of training, consulting, and programming expertise. **Reliability**: Embedded devices require high reliability. The absence of pointers and its automatic garbage collection eliminates some bugs and the risk of memory leaks. The language confines the use of memory and system resources, which contributes to greater reliability [15,16].

### 6. Implementation of a noise suppression system

Sweden one of the leader country in the world in manufacturing suppression system and to export it to the whole world. Many people, around 65000 inhabitants have problem with hearing [20] especially hearing injuries. The Embedded warning system is done to minimize the noise suppression, to reduce hearing injuries in noisy environments and to mind hearing loss in humans.

#### 6.1 Background acoustic theory

Usually the manufacturing of acoustic embedded system unit adjusts by dosimeter unit to the criterion level. Many dosimeter give an equivalent sound or noise level, this is the average exposure
level for noise. If a person is exposed over an eight-hour work shift to varying noise level, it is possible to calculate an equivalent sound level which would equal to the same total energy exposure. Figure 8 indicates the 82dB in sound presser level meets with around 1.2 KHZ in an octave band, that the maximum frequency to humans ear under injury level [20].

![Image](image.png)

**Figure 8: The acoustic graph frequency**

6.2 What noise suppression system

A suppression noise system is a system done to reduce the hearing injuries. By sending a signal in term of vibrator (alarm) once the noise exceeded more than 82dB. When the noisy environment reduces to less than 82 db, then the signal will be off or the vibrators switch off. The main parts of noise suppression system are the calculations of the sound presser and sound power as the following:

6.2.1 Sound presser calculation (SPL)

The intensity of a sound wave is measured by the square root of the time-averaged value of the square of the pressure amplitude i.e. \( \sqrt{(p - po)^2} \) But it is the convention of sound specialists to use a logarithmic scale of sound intensity whose unit is the decibel (dB), and having a zero value \( \sqrt{(p - po)^2} = 2E(-5)Pa. \) on this scale the sound pressure level (SPL), measured in dB, has the value:

\[
SPL = 20 \log \left( \frac{\sqrt{(p - po)^2}}{2 \ E (- 5) \ pa} \right) \text{ dB}
\]

The reference pressure level of 2E (-5) Pa for zero dB on the sound pressure level scale was chosen because it is the approximate amplitude at which a sound wave is barely audible to the normal human ear. For each factor of ten increases in pressure amplitude above this level, the sound pressure level increases by 20dB. Continuous exposure to sound pressure levels above 82dB (i.e., (p-po)2 = 0.60 Pa) can cause hearing loss in humans [20].

6.2.2 Sound-power-calculation (Lw)

The sound power level (Lw) is often quoted on machinery to indicate the total sound energy radiated per second. It is quoted in decibels with respect to the reference power level. The reference level is 1 Pico-watt. One watt of radiated sound power is represented as “Lw=120dB re on Pico-watt” if the reported sound power is in term of A-weighted. The sound presser level (SPL) resulting from sound power level (Lw) being radiated into free space, e.g. over a paved surface, is computed from:

\[
\text{SPL}=Lw -20\log (R) -11 \text{ dB re } 20 \mu Pa \text{ (R in meter)}
\]

\[
\text{SPL}=Lw -20\log (r) -0.7 \text{ dB re } 20 \mu Pa \text{ (r in feet)}
\]

6.3 Embedded Java applications

The Application of embedded Java are:

1. New projects which involve multiple levels of hardware and software. For example, an embedded system, a PC or local server gateway, and backend processing. Here Java is a huge win, since each team can share the same tools and APIs. Very difficult in C language.

2. Systems with TCP/IP. Java has a consistent set of proven and stable APIs for networking, and has had from day one. No other language can claim that. (about C is the plethora of non-standard, arbitrary add-on packages for serial and network I/O).

3. Systems where network security is important. Java has a very robust, mature security model which is has been pounded on by a huge community.

4. Systems which will be maintained and updated frequently. Most of the total lifecycle cost of a hardware/software system is in maintenance, not in the initial development. Java is a clear winner here because of better built in documentation, standard libraries, built in exception handling, and programmer efficiency.

5. Systems where exception and error handling is critical. Java has a good deal of enforced exception and error handling.

6. Systems where runtime reliability is important. You are thinking "where is this *not* the case? But C has essentially no runtime error checking, memory allocation is all manual, etc. Java does memory management automatically; bounds-checks array access, etc so it handles the big problems.

7. Fast system development and prototyping. You can get a lot done in a short amount of time with Java because it has such a rich library of functions already built into the language. There is a huge
amount of open source, free software which you can probably leverage -- and most of the time it will actually compile and run on *your* hardware!

But Java's large code size, slow execution time and under termination caused by garbage collection prevented it from using widely in the embedded systems. Combining JVM and JIT or using Java processor or speeding synchronization monitor to speed Java execution, using software approach or hardware approach to solve garbage collection problem in real-time system. Embedded system time limits is very important “response time” that means all specified dead lines must be met. They are always involves both the software and hardware co-development and often easier understood in term of Smart Devices, Intelligent, or Automated units. The scheduler (dispatcher) is the main part of the kernel responsible for determining writing task will run next in operating systems. Most embedded system kernels are priority based. The priority for each task is application specific. The deadline is a constraint on the latest time at which the operation must complete. The Java network (JN) is a soft-real-time kernel because the JVM has no hard-real-time requirements. Therefore, JN uses classic software architecture for a soft-real-time kernel. The architecture has been used by the successful line of kernels implementations. The kernel architecture can be considered a Serially-Reusable Interrupt-Extension. Some of its characteristics include [5,9-14]:

1. The kernel can be thought of as a single task that runs as an interrupt-enabled follow-on extension to all interrupt routines. The kernel runs from start to completion every invocation. Although the kernel can be interrupted, the kernel cannot block, and kernel CPU state is not saved across kernel invocations. Rather, the kernel always starts at a known code and stack location. The kernel uses a single kernel stack (on a uniprocessor).

2. The kernel consists of a single work-loop driven by a queue of control blocks which form a real-time subroutine or co-routine dispatch scheduler. Each control block contains the address of a routine which the kernel must execute. Historically, these routines have been called fork routines in the Cutler kernels. Other common names for such routines, especially as found in I/O managers, include Second Level Interrupt Handler and Deferred Procedure (DPC). Since fork routines cannot block and must have a bounded execution time, they have many characteristics of routines written for a hard-real-time environment. Neither the fork routines nor the kernel need to use explicit mutual exclusion to access global data structures (on a uniprocessor), reducing the need for explicit synchronization.

3. The kernel is activated by an interrupt, either hardware or software. If an interrupt occurs when the kernel is not active, the kernel is started at the end of the interrupt routine, typically after the interrupt routine queues a control block designating a fork routine to handle the events triggered by the interrupt. If the kernel is active when an interrupt occurs, the interrupt routine simply queues a control block for follow-on processing and returns from the interrupt back into the kernel.

4. After executing all routines corresponding to control blocks, the process scheduler is run if a global flag indicates that one of the routines has altered the scheduling status of a process. The selected process is then dispatched. Synchronisation mechanisms are objects used to control the execution of concurrent threads, so that there is no conflicting usage of shared resource or ineffective resource usage. Developing for object oriented modelling embedded systems is one of the most technologically challenging systems. Challenges are to be highly event-driven, concurrent, and often distributed. Stringent requirements must be met for latency, throughput, and dependability. Capturing and effectively communicating designs for real-time systems it is a great step forward.

7. Embedded-system-in-C

Embedded-C usually means VHDL and/or MicroC/OS-II Compilers which are complete cross-compiler solutions that include a compiler, assembler, and linker, all optimized for building embedded system applications. MicroC/OS-II Compilers might use as stand-alone tools. In addition, they are compatible with many third-party debuggers, emulators, and real-time operating systems [8,9].

- Advanced C/C++ optimizations dramatically reduce application memory usage
- Pre-compiled header files slash compile times
- Flexible code and data segmentation enables easy ROMing of applications
- C language extensions minimize the need for assembly language.
- Compliance with ANSI C and C++ standards

MicroC/OS-II (Meets the requirements of safety-critical systems and the most widely used real-time kernel in the industry) is a highly portable, ROMable, scalable, pre-emptive real-time, multitasking kernel (RTOS) for microprocessors
and microcontrollers with performance comparable to and in some cases exceeding, commercially available kernels. Written in ANSI C for maximum portability, MicroC/OS-II has been ported to more than 40 different processor architectures ranging from 8 to 64-bit CPUs. Since 1992, MicroC/OS has been used by thousands of developers worldwide in applications ranging from cameras to avionics. Certified for use in safety-critical systems, this RTOS has proven to be robust, reliable, and safe enough for applications. MicroC/OS-II describes the design and implementation of MicroC/OS-II, teaches the fundamentals of a multitasking real-time system, kernel structure, scheduling, context switching, task, time management, intertask communications and synchronization, dynamic memory allocation.

A real-time system is one in which timeliness is important; it is reactive, has concurrently executing threads, and is not deterministic. To specify a model for a real-time system, time requirements, concurrency, asynchronous communication, and synchronization must be considered. A real-time system is often embedded, meaning that the software is tightly integrated with specialized hardware. Embedded systems can be categorized as hard real-time or soft real-time. A hard real-time system is one in which a failure could mean loss of life; thus fault tolerance is of the utmost importance. Real time and object orientation are integrated through the definition of active classes and objects. An active class has its own thread of control and can initiate actions by itself. That is in contrast to "normal" passive objects, which execute only when some other object performs some operation on it and passes back the control when the operation returns. Active classes and objects are implemented through processes or threads supported by the operating system. Active objects communicate either through ordinary operation calls (called synchronous messages in the object-oriented world) or through specialized mechanisms such as mailboxes or message queues in which messages can be delivered asynchronously. An asynchronous message means that the sender doesn't wait for the receiver to be ready to accept the message, nor does it automatically wait for any return result. The behaviour of active objects is modelled using events; an asynchronous message signal is one type of event. Other events are that a guard-condition becomes true, a passage of time elapses, or a synchronous message is received from another object. Synchronization mechanisms are things such as semaphores or critical regions that protect a specific code area, which in turn protects some shared resource such as a passive object or a device. Synchronization mechanisms are used to prevent problems such as conflicts in parallel usage of shared objects, deadlocks, or just inefficient resource usage of the processor caused by having threads executing in endless "busy wait" loops. To define a method/process for object-oriented real-time systems, these aspects must be addressed: determine a basic model for separating processes/threads from objects; identify and define active classes, communication, and synchronization of the active classes; and, finally, design a suitable mapping to an implementation environment. Although there are many methods on the market today for embedded systems, there is still a lack of methodology for combining embedded systems and object orientation.

### 7.1 Embedded C applications

1. Small, fixed purpose, simple, high-volume, price-sensitive systems.
2. Systems which will never be maintained and updated. This generally means simple fixed purpose systems. In such systems the overhead of Java may not be worth the benefit.
3. The PalmOS and its development environment is mature and stable. Why re-write it all in Java?
4. I do not know about any Java for DSP processors, in DSP hardware (many questions asked).
5. Extremely high speed control systems. In case of using a fast bit slice or RISC chip with hand coding (by experienced Programmer) and barely meeting the requirements, it needs a high level language.
6. On a complex and already late project written in C or assembly. This is not the time to be learning a new tool. There are millions of kinds of embedded systems, so it's not really possible to say what the "average" or "normal" embedded system is like. Everything is different. Most embedded systems are still programmed in assembly language, especially small ones (8-bit or 16-bit CPUs). Most of the rest are programmed in C, especially for 16-bit and 32-bit CPUs. Very few systems are programmed in Java or any other language. C and assembly are the most common languages. This is because C and assembly produce very efficient, tight, fast, and small code. ROM, RAM, flash, and disk are usually small and expensive resources. C is good at this but Java is not well at this. Java produces very big and very slow programs. Java requires an enormous amount of memory. Java was designed for computer systems, not small embedded systems. There are some versions of Java modified for embedded systems, but these are still very large and very slow, and they do not really implement all of Java, only parts of it. Java is well for Web browsers, PCs but it's not really a good choice for most embedded systems, in my opinion.

### 8. Result
The implementation of the suppression of a noise system has implemented in C language and Java/
hardware design language (HDL) such as Verilog, any more [12]. They generally use a high-level programming. Hardware do not draw schematics as FPGA is looking more and more like especially for ASICs and other custom chips such of choice for some more time. Hardware design - MicroC/OS-II would be the programming language least for really small embedded systems, The C or 16 would be substantial. Thus, it is anticipated that, at the JVM to work on obscure or obsolete chips constraints of the required system and of modifying developing a JVM subset which would fit the embedded system, because the overhead of Java makes it proper for embedded systems. But a lot of efforts should be done specifically those efforts of improving Java in a whole. But current it's not ready for the embedded system. Embedded systems are difficult things to work with. They are small and cheap and they have extremely limited system resources to compensate. More specifically, while designing for an embedded system I have to consider one or more of the following constraints:

- Obscure special purpose or low performance general purpose CPU.
- Limited RAM, in the low hundred kilobytes.
- No hard disk. All permanent storage must be in ROM.
- Limited display capabilities.

In the view of these limitations, it would be very difficult to write a Java-based service on an embedded system, because the overhead of developing a JVM subset which would fit the constraints of the required system and of modifying the JVM to work on obscure or obsolete chips would be substantial. Thus, it is anticipated that, at least for really small embedded systems, The C or MicroC/OS-II would be the programming language of choice for some more time. Hardware design especially for ASICS and other custom chips such as FPGA is looking more and more like programming. Hardware do not draw schematics any more [12]. They generally use a high-level hardware design language (HDL) such as Verilog, VHDL or µC/OS-II (micro C/OS-II). A few lines of VHDL code looks a lot like a C listing at first glance later never know that the language was describing hardware instead of the control flow of a program [18,19]. And that is exactly my point. In the hardware world, the de facto standards of VHDL, Verilog and µC/OS-II are creaking under the weight of new 10-million-gate chips with multiple embedded processors and more computing power [12]. It's not an efficient use of the engineers' time to design a chip that big, gate by carefully crafted gate. Some faster and more productive method is required. Obviously, the opinions about Java are much divided [13]. Some believe it's the next BIG thing, others argue that it's inadequate for embedded systems. I don't see how Java addresses better embedded needs (e.g., I see state machines, or design-by-contract much more important than anything that Java has to offer). I also find the Java threading model (based on mutual exclusion and blocking the synchronized methods) inadequate for hard real-time applications. The Advantages of Java are:

- Java is good for module reuse.
- It is easy to produce bug-free software using Java.
- Using Java will reduce problems and cost caused by code maintenance.
- Application written in Java is platform-independent.
- Java is good for networked software (dynamic loading).
- The interpreted bytecode has high density, which increases the bandwidth effectively.

Although Java has so many advantages, people working on embedded systems still hesitate to use it, because Java has main disadvantages for real-time systems such as %size is usually large, speed is relatively low and undermining caused by garbage collection. How to overcome these disadvantages while exploring the benefits of Java, and finally to bring Java into real-time system. It is a huge research area at these days. Numbers of research work has been done in this area. The Javelin Stamp since it is not real Java; it has one data type, it is not standard Java. It is too limited. It has no threading, garbage collection, and will not support standard Java packages, for example, serial I/O. Embedded Programmer must develop code using custom IDE; the embedded programmer cannot use standard Java tools. The C language is written for embedded and real-time system and other mixed small application systems. The C has no problem with memory handle and serial I/O. Java stamp is probably not fast enough to record the sound directly in this work.
10. Conclusion
Current Java is not good enough for being widely used in the embedded system, specifically in the hard-time embedded system. Although a lot of research works have been done in improving Java from different aspects, no integrating ways to improve the Java on the whole. Java is still not ready enough for widely used in embedded system. It does work when choosing a tailored Java according to the requirements. It’s possible to obtain the good performance if case of understanding well the Java, the systems requirements and the possible environment in which Java will run. There are some successful cases of using Java in the embedded system. There is a lot of confusion these days over whether Java will be a compiled or interpretative language - or both. An interpreted embedded implementation brings many benefits - small code and the possibility of Forth-like interactive development. A compiled version will run faster, almost a requirement for Java's success in real time applications. Java's current implementations leave some serious problems for embedded systems. Apparently it is garbage collection scheme, once started, must run till completion. C language took many years to be commonly used in the embedded world. The reason is simple: until a large number of C programmers were available, no one was willing to risk their next product using this new and risky technology of Java. A question raised in my mind whether Java succeed or not? The potential is exciting. Java could be really interesting for tiny applications. If coupled with an interactive development environment, perhaps using flash memories, small systems could do in no time at all. I see C or C++ is more adequate language than other language in small embedded system programming. Java/Java stamps and Javelin are convenience in computer systems but not in small embedded systems.

11. Future work
Although there are many methods on the market today for embedded system, there is still a lack of methodology for combining real-time system and other object orientation programming technique. Many studies still to explore in embedded system both in micro/OS-II, Java/Javelin and/or Java stamp to fillful the need of the market. The cost of transistors is approaching zero while the cost of engineering talent is higher requested. Special research and studies in Javelin/Java stamps still a big question.

12. Acknowledgements
Thanks are given to Steven Kirk for his assistance and for proof reading this manuscript. Thanks are also given to Christfer Selverfors for his helpful assistance to obtaining the Javelin board regarding the subject. Thanks are also given Samantha Jenkins for her helpful articles. Thanks also giving to Andy Lisley for supporting in Berkeley University and to Bruce Boyes in Utah University for supporting of information.

13. References

