IST Programme

IST-1999-12504

AJACS

Applying Java to Automotive Control Systems

"AJACS Concluding Paper"

Technical Paper
Version 1.0

Editors : Xavier Cornu, Antonio Kung. Trialog

July 2002

Table of Contents

1. INTRODUCTION 3

1.1 SCOPE AND EXPECTED READERS 3
1.2 CONTEXT OF AJACS PROJECT 3
1.3 CONTENT OF DOCUMENT 4
1.4 GLOSSARY 4
1.5 ACKNOWLEDGEMENT 5

2. EMBEDDED SYSTEMS 6

2.1 INTRODUCTION 6
2.2 THE CASE OF AUTOMOTIVE CONTROL APPLICATIONS 7
2.2.1 Automotive Control Systems Today 7
2.2.2 The OSEK/VDX Initiative 9

3. PROGRAMMING LANGUAGES 11

3.1 THE PROGRAMMING LANGUAGE C 11
3.2 MAIN CHARACTERISTICS OF OBJECT ORIENTED LANGUAGES 11
3.3 THE CASE OF JAVA 11
3.4 APPLYING JAVA TO REAL-TIME 12
3.4.1 The Real-Time Core 12
3.4.2 Sun Real-Time 12

4. JAVA FOR REAL-TIME SYSTEMS 13

4.1 JAVA FOR DEVELOPMENT OF EMBEDDED SYSTEMS 13
4.1.1 Principles 13
4.1.2 Intended Use in Embedded Systems 15
4.1.3 Recommendations 19
4.2 REAL-TIME SUPPORT 19
4.2.1 Principles 19
4.2.2 Intended Use in Embedded Systems 22
4.2.3 Recommendations 28
4.3 SUPPORT OF EXCEPTION 29
4.3.1 Principles 29
4.3.2 Use In Embedded Systems 36
4.3.3 Recommendations 38
4.4 INITIALISATION 39
4.4.1 Principles 39
4.4.2 Intended Use in Embedded Systems 44
4.4.3 Recommendations 50
4.5 MEMORY MANAGEMENT 50
4.5.1 Principles 50
4.5.2 Intended Use in Embedded Systems 56
4.5.3 Recommendations 57
4.6 NATIVE INTERFACE 58
4.6.1 Principles 58
4.6.2 Intended Use in Embedded Systems 64
4.6.3 Recommendations 71

5. LESSONS LEARNED, RECOMMENDATIONS 73

6. REFERENCES 75

1. Introduction

1.1 Scope and Expected Readers

The objective of this document is to provide a technical analysis of the issues of using Java[1] for embedded real-time systems, with a particular focus on embedded static systems such as automotive electronic control unit and to provide a set of recommendations.

Because this document is intended for two different types readers : Java programmers who get involved into Embedded Systems programming, and Embedded systems programmers who are moving to the Java programming language, we have decided to include in the document suitable introduction sections on either Java or on real-time systems.

1.2 Context of AJACS project

The success of Java in the industry, and the increasing interest of the real-time community to use it, led this consortium to consider an investigation on the use of Java technology in automotive control applications :

- Through its object-orientation, Java makes it easy to design software component with strictly defined interfaces, even at the source code level. Methodologies like OMT and UML can be adopted with little effort.
- The Java language is designed to be platform independent. Thus hardware independence and portability requirements are largely fulfilled.
- Easiness and robustness attributes contribute to high programmer productivity and low defect rates. This will help the development team focus on high-level activities like component-based design.

To this end, the AJACS initiative[2] was started in February 2000 with the following objectives:

- Define an open technology which relies on existing standards of the automotive industry, such as OSEK/VDX.
- Retain the benefits expected from object-oriented language programming in terms of software structuring, reusability, dependability in particular retain the portability, and robustness attributes associated with Java.
- Address technical issues created by drawbacks of the Java language in terms of real-time and determinism support for embedded systems with high integrity constraints. In the case of automotive control, this means supporting the same kind of real-time constraints which non-Java based ECUs currently handle and targetting the type of memory footprints that are acceptable in the automotive industry (e.g. 256 Kbytes ROM, 16 Kbytes RAM).

---

1 Java is a trademark of Sun Microsystems in the United States and in other countries

2 AJACS includes Trialog, PSA, Centro Fiat Riserche, Mecel AB, U.Karlsruhe/IPD. AJACS is partially funded by the European Commission.

1.3 Content of Document

This document includes the following sections:

- This introduction section
- An introduction section on embedded systems and on automotive control systems
- An introduction section on programming languages
- An analysis section covering the following topics :
  - Java for development of embedded systems
  - Real-time support
  - Support of exception
  - Initialization
  - Memory management
  - Native interface

  Each of these topics is treated in terms of principles, intended use in embedded systems, and recommendations
- Lessons learned
- References

1.4 Glossary

API	Application Programming Interface
CPU	Central Processing Unit
ECU	Electronic Control Unit
HIP	High Integrity Profile
HIPA	High Integrity Profile for Automotive control
JMM	Java Memory Model
JVM	Java Virtual Machine
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeug (Open systems and the corresponding interfaces for automotive electronics) / Vehicle Distributed eXecutive.
OMT	Object Modeling Technique
ROM	Read-Only Memory
RTOS	Real-Time Operating System
UML	Unified Modeling Language

1.5 Acknowledgement

AJACS was partially funded by the European Commission.

We would like to thank in particular Tom Clausen, project officer of the Commission as well as Alain Perbost from Thomson-CSF and Karl-Heinz Krause from Siemens. They provided us with many valuable comments and suggestions.

We would also like to thank the J consortium which provided us with many technical support in AJACS work towards a high integrity profile specification for automotive control applications. Kelvin Nilsen from Newmonics, chairman of the Technical committee of the J consortium reviewed an initial version of the document and provided many invaluable comments. We have included most of his comments in this report.

We finally would like to thank the OSEK-VDX committee which provided us with guidance on how the AJACS project should proceed in terms of standardization.

2. Embedded Systems

2.1 Introduction

Embedded systems can be defined as follows [Gupta02]

> *"An embedded system employs a combination of hardware & software (a "computational engine") to perform a specific function; is part of a larger system that may not be a "computer"; works in a reactive and time-constrained environment. Software is used for providing features and flexibility. Hardware (e.g. Processors, ASICs, Memory,...} is used for performance and sometimes security."*

Embedded systems exhibit the following typical characteristics [Gupta02]:

- *they perform a single or tightly knit set of functions (they are not usually "general purpose"),*
- *they are increasingly high-performance & real-time constrained,*
- *power, cost and reliability are often important attributes that influence design"*

Finally embedded systems are very diverse.[Gupta02] provide the following examples :

- *a pocket remote control RF transmitter would have the following specific attributes*
  - *100 KIPS, crush-proof, long battery life*
  - *Software optimized for size*
- *An industrial equipment controller would have the following specific attributes*
  - *1 MIPS, safety-critical, 1 MB memory*
  - *Software control loops*
- *A military signal processing system would have the following specific attiibutes*
  - *1 GFLOPS, 1 GB/sec IO, 32 MB*

Automotive electronics can exhibit the same diversity from engine control, ABS, dashboards which are very constrained to navigation system which would include several megabytes memory.

Embedded systems are also sometimes categorized according to whether they are closed/static systems versus open/dynamic systems. In closed/static systems, all computing resources are predetermined (in terms of threads, memory, and so forth). This is often a characteristics or even a requirement of high integrity applications (such as avionics, automotive or space applications). Closed/Static systems allow for memory footprint optimization and therefore cost optimization for two reasons :

- because all memory resources are predetermined, no extra memory space is needed
- because all entities are predetermined, significant amount of data can be put in ROM instead of RAM. ROM is much cheaper than RAM : a typical single chip configuration could contain 64kbytes ROM and 1kbyte RAM.

Typically used programming languages today for the development of embedded systems are

- Assembler. Still many embedded systems are still written in assembler, not for lack of computing experience and know-how, but mainly for cost reasons. In many applications where software complexity is still small, sticking to assembler allow the use of cheaper components
- C. Most automotive control electronics are now written in C. The change took place about 10 years ago when the complexity of such systems increased significantly.
- Ada is used in many complex embedded systems such as military applications, avionics and space applications. Ariane launcher software in written in Ada.

2.2 The Case of Automotive Control Applications

2.2.1 Automotive Control Systems Today

Today vehicles include an increasing number of electronics systems. It has been estimated by Dataquest that the average semiconductor content of a vehicle will reach $240 by 2001, with consumption of DSPs, microcontrollers and microprocessor reaching $4.9 billion. ***Electronic control units*** or ECUs now play a crucial part in all the functional areas of a vehicle such as infotainment / multimedia (e.g. radio system, road guidance, cellular phone), body control (e.g. instrument panel, window lift, automatic door lock), or vehicle control (e.g. engine management, transmission, brakes).

This was made possible through the advent of networking technology such as CAN. Typical vehicles consist of several interconnected networks that more or less reflect distinct functional areas. For instance, the Volvo S80 includes «18 ECUs connected via six networks: a low-speed body electronics CAN bus (125kbit/s), a high-speed powertrain CAN bus (250kbit/s) and four other networks».

ECUs are typically designed and developed by OEMs according to requirements set up by car manufacturers. Until recently, each ECU was dedicated to a single user-function (e.g. climate control) and OEMs had entire freedom for the implementation (hardware and software). Car manufacturers now try to introduce more flexible development processes that support the breakdown of the user function into fine-grain sub-functions with very precise requirements, in particular at the software level. Two examples of such breakdown (“Display speed” and “Cruise Control” user functions) are presented in Figure 1. At some point in the development process, the resulting sub-functions will be mapped to the hardware subsystems, the ECUs, that form the physical architecture (see Figure 2). The intent of the car manufacturer is that the functional specification and the physical architecture design tasks can be carried on in parallel with definite synchronization points (the tentative physical allocations of sub-functions).

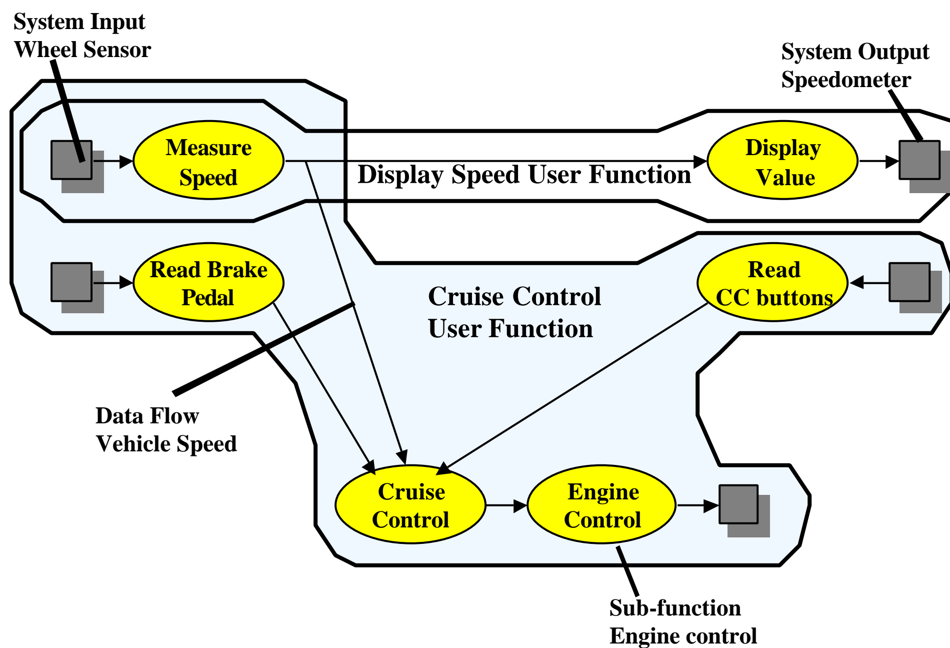


Figure 1 - Example of User Functions

The car manufacturers expect many benefits of such a development process : having more control on the specification of the user functions, protecting trade secrets by keeping some crucial sub-functions under their only control, reducing development duration and reducing the resulting bill of materials for the vehicle electronics.

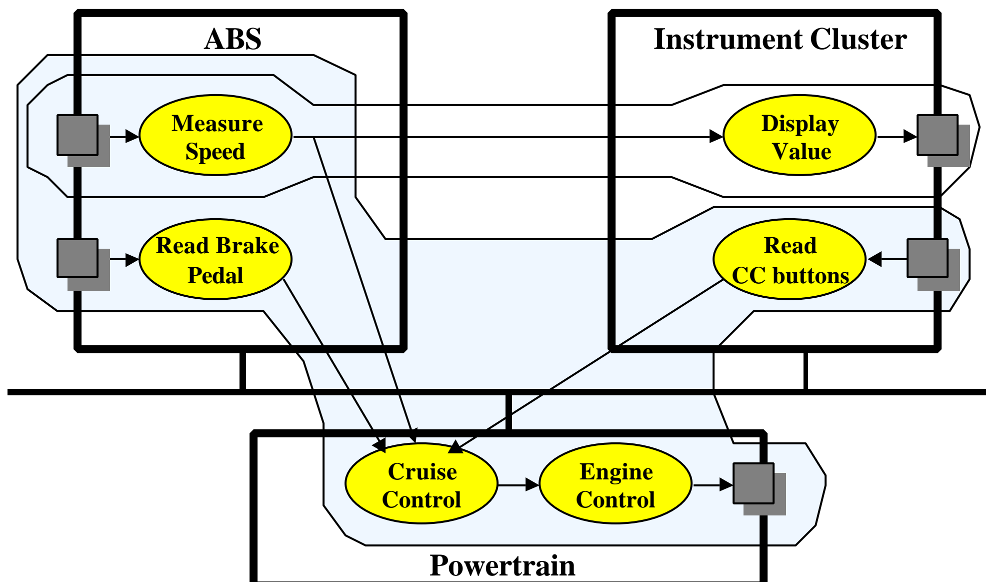


Figure 2 - Example of Physical Allocation

Resulting from this trend, the relationship between car manufacturers and OEMs is thus starting to change. OEMs may be subcontracted for the development of software components only

(e.g. an entire user function, or all the sub-functions sitting in the engine control ECU). Or OEMs may be subcontracted for the provision of an "incomplete ECU", that is an hardware and software platform on which sub-functions can be further added by the car manufacturer or some third-party (e.g. the engine control ECU without the sub-functions related to automatic gear shift).

From a different perspective, the OEMs also seek for a more flexible development process. Many of them try to set up generic hardware and/or software platforms that would fulfill all the general requirements for a particular functional domain (e.g. climate control) ; the intent is that a large range of products can be easily and cheaply derived from these generic platforms by a quick tailoring to the specific requirements of each individual project. At the software level, such generic platforms usually rely on a definite set of fine-grained components that can be interconnected in various ways, depending on the actual requirements to be fulfilled.

To address the trend towards advance electronic architectures, the whole industry has identified the need to provide guidance on the transition towards advanced electronic architectures. It has been pushing from the start for the definition of open systems, with the definition of corresponding interfaces (APIs) as in the OSEK/VDX (1) initiative presented below. It has also been pushing for the use of advanced software engineering methods (such as OMT, UML), and approaches promoting software reuse (such as object-oriented programming).

2.2.2 The OSEK/VDX Initiative

OSEK, an abbreviation for the German term "Offene Systeme und deren Schnittsellen für die Elektronik in Kraftfahrzeug"[3], is a joint project started in 1993 by the German automotive industry. Initial project partners were BMW, Bosch, Daimler-Benz, Opel, Siemens, Volkswagen and the IIIT of the University of Karlsruhe as co-ordinator. French car manufacturers Peugeot and Renault joined OSEK in 1994 introducing their VDX-approach ("Vehicle Distributed Executive") which was a similar project within the French automotive industry. The first results of the harmonization effort were presented by the ***OSEK/VDX*** group in 1995.

An ISO submission process started in early 2000 for the latest release of the specifications. As of today, OSEK/VDX is probably the most successful undertaking concerning standardization in the embedded software industry: there are more than 10 different providers of OSEK/VDX components and many more proprietary implementations from car manufacturers and OEMs.

The scope of the OSEK/VDX specifications includes :

- ***OSEK/OS***, a specification of behavior and APIs for a real-time operating system that makes provision for the execution and synchronization of application tasks and interrupt routines.
- ***OSEK/COM***, a specification of APIs and network protocols that supports exchange of

[3] English translation is "Open Systems and the Corresponding Interfaces for Automotive Electronics".

data messages between software components, either through local communication (i.e. the transmitter and the receiver are located on the same ECU) or through a network.

- ***OSEK/NM***, a specification of APIs and protocols that serve as a basis for network wide negotiated management functions. Basically, it supports a distributed monitoring scheme for the ECUs, thus allowing a particular sub-function to be informed about the current availability of the other sub-functions it relies on.
- ***OSEK/OIL***, a language to describe the configuration of a particular software component or of a complete application. This configuration includes for instance the definition of the operating system tasks, as well as the description of the messages to be received and/or transmitted.

OSEK/VDX provides a standard architecture for distributed control units in vehicles. It meets the two stringent automotive requirements we have previously mentioned: real-time support and small footprints. This is reflected in OSEK/VDX characteristics :

- It is a static system. All entities are known and declared in advance. Furthermore, all data structures within OSEK/VDX are defined and initialized statically using the OIL language. For instance a task descriptor contains information such as a starting address or priority which are static. Such information is stored in ROM and not in RAM. In order to bring some flexibility, the notion of mode management is also defined. Applications may switch from one mode to another. This switch involves the initialization of the whole system to another set of predefined objects.
- Many OSEK/VDX-based applications are interrupt intensive applications. As a matter of fact, some of them could be entirely interrupt-oriented. This is why OSEK has defined the notion of basic tasks or entities which can be considered as logical interrupts.
- Many applications need to react to cyclic events, although not periodic. This is the case of engine control. OSEK provides services for associating events with a flexible notion of counters, which could be a timer counter or any type of counter (e.g. an event from a sensor).
- Because of the wide range of applications to be supported, the technology offers a number of options: basic tasks (logical interrupt routines) or extended tasks (standard tasks with waiting states), preemption or not, multiple activation or not of the same task.
- A common mutual exclusion service with PCP (Priority Ceiling Protocol) is supported for basic tasks and extended tasks.
- Extended tasks may wait for events. Events are thus associated with configurations of extended tasks.

3. Programming Languages

3.1 The Programming Language C

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the type-less language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today.

Besides its high level type system, it benefits from its low level operators. This allowed programmers to implement algorithms in a very efficient way and still keep the algorithms in a portable format. UNIX systems and its tools are written in C.

Especially the possibility to write low level programs, which can access the hardware directly, open the door to embedded systems.

A normal C programmer focuses on the implementation of algorithms and the control-flow of the program.

3.2 Main Characteristics of Object Oriented Languages

In contrast to imperative languages, like C, the programmer focuses on data-structures and the implementation of methods which transform the data-structure in object-oriented languages. Furthermore the programmer has a closer look on the interactions and dependencies of his data-structures, e.g. inheritance. This introduces a new way to describe the algorithm and its data-structures. The algorithm is implemented by the combination of such data-transformation methods.

Such data-structures and methods are grouped in *classes*. An instance of every classes can be created and is called *object*. Classes can have a inheritance relation. If a class C inherits from a class D, then it inherits all is data-structures and its data-transformations (e.g. class Human inherits from class Mammal). But the new class can overwrite methods to adept the behavior of an instance

As a consequence, object-oriented languages have a very complex type system (a Human is a Mammal), which make a method call complex. If a method is called to transform some data-structure of an instance, then it could be possible that this instance is a subclass with a overwritten method. A so called *polymorphic method invocation* must be performed.

3.3 The Case of Java

The programming language Java is a combination of the former two points. It combines some syntactical properties of the language C with the features of an object-oriented language.

Besides this it claims to be: simple, distributed, interpreted, robust, secure, architectural neutral, portable, high-performant, multi-threaded and dynamic.

Most such claims are guaranteed by the definition of a Java Virtual Machine (JVM). A program, which is written in Java, shall behave on every platform in the same way, always. Every Java Program is compiled into Java Byte Code, which is interpreted and executed by the JVM.

3.4 Applying Java to Real-Time

3.4.1 The Real-Time Core

The Real-Time Core, created by members of the J-Consortium, defines a platform for the use of Java in a real-time environment [Real-Time Core Specification]. The main issue of the Real-Time Core definition is to guarantee the portability of Core application software and to ensure the compatibility between implementations of Core developments tools and run-time environments.

To distinguish between the standard Java and the real-time variant the former is called *baseline* Java and the latter one is called *Core*, in short. Both variants exists simultaneously. But the real-time variant's root-class is not the class *Object*, but the class *CoreObject*. Therefore, development tools like compilers have to take case of this fact by replacing the class *Object* by the class *CoreObject* (root-class replacement).

Furthermore it defines an environment in which a core application runs. To this environment belongs the execution model, the memory model and a core-class hierarchy.

3.4.2 Sun Real-Time

The Sun Real-Time specification is another official specification for developing real-time applications with the Java platform [RTJS]. The Real-Time Specification for Java (RTSJ) is a reference to the semantics, extensions, and modifications to the Java programming language.

The RTSJ shall enable the creation, verification, analysis, execution, and management of code written for the Java platform for which the correctness conditions, timeliness, and execution predictability are paramount.

This specification provides programmers with the ability to model applications and program logic that require predictable execution which meets hard real-time constraints.

4. Java for Real-Time Systems

This section covers the following topics :

- Java for development of embedded systems
- Real-time support
- Support of exception
- Initialization
- Memory management
- Native interface

For each of these topics, we have included the following sections :

- A section explaining the principse
- The intended use in embedded systems. This section will explain the issues and also list possible solutions
- A recommendation section

4.1 Java for Development of Embedded Systems

4.1.1 Principles

4.1.1.1 Java Development environment

Root Classes and Standard Libraries

As an object oriented language, all programming entities are described as classes and objects (i.e. instantiation of classes). There is in the language a minimal set of "root" classes. These classes are defined in the `java.lang` package : `Object`, `Class`, `String` and `Throwable`. These classes are called "root" classes because they are the smallest subset of Java classes that can exist without dependencies. Each of the root classes depend on other root classes, but not of any other class.

As most programming environments, Java Virtual Machines are not delivered with the support of root classes only. A set of libraries (a.k.a. packages in Java), that helps the programmer perform basic operations such as printing functions, thread management, basic exception classes and so forth is also provided.

Even though JVMs are developed by competitors, the most frequent set of library used is from Sun Microsystems. There are at least two reasons for this : first the investment to redevelop standard libraries is important. Second implementing its own library could lead to incompatibilities.

When Sun microsystems library is used, then this will involve a licensing agreement with the JVM developer whereby the JVM developer is not allowed to change the APIs. The rationale behind this constraint is to ensure portability of Java applications.

Pre-processing

Many programming languages and especially C have a pre-processing phase which allow programmers to :

- define constants
- define macros
- define conditional compilation of source file blocks

Java has no such facilities.

Remarks from Kelvin Nilsen : while Java has no such facilities, capabilities for (a) defining constants, (b) defining macros, and (c) defining conditional compilation of source blocks can be provided by typical Java implementations. In particular,

- a constant can be defined by using the final qualifier in the declaration, as in:

```
final boolean Verbose_Debug_Info = true;
```

- Most of what a C programmer would want to accomplish with macros can be achieved by writing static or final methods. Almost all compilers will in-line the implementation of these methods if they are sufficiently small, as in:

```
static void DebugOutput(java.lang.String msg) {

  if (Verbose_Debug_Info)

    System.out.println("DEBUG: " + msg);

}
```

- Java compilers are good enough to conditionally compile code that is easily shown to be dead at compile time. Given the two bodies of Java code shown above, the following line translates into zero machine code:

```
DebugOutput("trace message, value of X: " + X);
```

Debugging

Debugging facilities for Java today are mostly for execution on host systems (PCs, or work stations). For instance Sun Microsystems JVM implementation comes with a debugger tool, called `jdb`. It allows the programmer to debug Java application either locally on the host or remotely through socket communication.

4.1.1.2 System Programming

Java was not intended for system programming.

4.1.2 Intended Use in Embedded Systems

4.1.2.1 Development environment

Root Classes

[Real-time Core Specification] explains that the semantics of the root classes as defined in the Java language cannot be kept as they do not meet requirements of real-time systems. For example, some services like `wait()` and `notify()` have to be redefined in original `Object` class: they rely on Java thread model which is not suitable for Real-Time.

Several approaches have been identified :

- Change the semantics of the root classes. This approach is suited to embedded systems which only contain classes and objects that are real-time
- Define new root classes (`CoreObject`, `CoreClass`, `CoreString` and `CoreThrowable`) which coexist with (`Object`, `Class`, `String` and `Throwable`). Therefore the real-time part of an application should use core classes while the non real-time part of an application can stick to standard root classes. The rationale behind is that by supporting both new root classes and standard root classes, it is possible to reuse standard Java applications. Having new additional root classes raise however a major technical issue : all classes inherit from `Object` not from `CoreObject`. [Real-time Core Specification] has proposed a "root replacement" mechanism whereby `Object`, `Class`, `String` and `Throwable` are actually replaced by `CoreObject`, `CoreClass`, `CoreString` and `CoreThrowable` respectively. A tool could be used to replace the classes somewhere in the compilation chain. The compiler front-end could perform internally all the substitutions of the root classes.

Standard Libraries

The standard library contains hundred of classes. Even though only needed classes are loaded, in practice, using one Java class in the standard library often implies using all the library. This is because standard Java classes may themselves use other classes of the library. For instance if Sun microsystems standard library is used then using just one class would then mean that several hundred standard classes need to be downloaded. This is an issue in embedded systems in terms of memory footprints. This means that standard libraries are often discarded and replaced by target-dependent libraries.

Pre-processing

Not being able to define constants, macros (or to have controlled in-lining capabilities) or define conditional compilation of source file block is an issue in most embedded systems where low memory footprints is a constraint.

Debugging

Debugging in embedded systems is either done through a simulator (i.e. a software that simulate the instruction set of the target processor) or through and emulator (i.e. a hardware systems which replaces a target processor and executes instruction set in real-time). It is therefore necessary that debugging support be provided for such environments. In the case of emulators for embedded systems, there are either proprietary or semi-standard interfaces allowing emulators to provide source code level debugging. Such interfaces consists of map files with explain the correspondence between each line of source code and the target code.

Java development environments should support the following :

- Provision of Map files. If Java bytes code is interpreted, then there is a risk that map files cannot be used because they would map the target code to the source code of the JVM, not to the Java application
- Trace-ability :
  - If native compilers generate C, then the map files will provide the correspondence between target code and the generated C source code. Then tracing back to the Java source code is not straightforward.
  - If direct target code is generated, then there might still be traceability issues as some Java construct might generate complex patterns of code

4.1.2.2 System Programming

Lacking Features

Embedded systems applications imply significant amount of system programming. This is not obvious in Java because of the following :

- No constants (in the sense of C)
- No macros (unless the underlying Java compiler provides controlled in-lining capability)
- No unsigned types
- Lack of support for bit management. Below is a typical C example of the handling of events represented as bits. The counterpart in Java could yield significantly more code because of bit operators would have to be replaced by method calls.

```
/* Event Waiting */
WaitEvent (IgnitionOn | IgnitionOff | GearShift);

/* Event Testing */
EventMask received;
GetEvent (Task1, &received);
if (received & IgnitionOn) {…}

/* Event Setting */
SetEvent (Task2, IgnitionOn | ValueRequest);
```

- There is no support of interrupt routines. Extensions proposed are the following :
  - [Real-Time Core Specification] introduces the notion of Interrupt Service Routines through the `ISR_Task` class. When an interrupt is triggered, all the work() methods of `ISR_Task` objects registered as handlers for this interrupt are run one after the other (in a decreasing priority order).
  - [RTJS] introduces the notion of Asynchronous Events handlers with the `AsyncEventHandler`: when an Asynchronous Event occurs, all the `run()` methods of the registered `AsyncEventHandler` registered as handlers for the corresponding `AsyncEvent` object are called.

I/O Management

Lack of system programming capabilities will make it difficult to develop I/O based applications. I/O management in C typically consists in using bit structures. A 8-bit I/O port of the micro-controller is represented by a char variable placed at the address of the corresponding register. The code to write in such a variable looks as follows:

```
#define REG_FOO *((unsigned char *)0x12345678) // Address found in CPU
datasheet
REG_FOO = 42; // directly write the value in the register
```

The Java language does not provide a means to represent directly a physical entity (no notion of physical address), so the only possibility is to use the Java Native Interface: the programmer creates an accessor function to access the physical variable in the C code. This make the previous example look as follows in Java:

```
class Register8 {
     int address = 0x12345678; // beware of sign
     Register8(int addr) { this.address = address; }
     void write(byte value) { // beware of sign
           native_write (address, value);
     }
     static native void native_write(int address, byte value);
}
class Application {
     Register8 REG_FOO = new Register8(0x12345678);
     REG_FOO.write(42);
}
```

With the corresponding native function:

```
JNIEXPORT void JNICALL Java_Register8_native_write
     (JNIEnv *env,
     jint address,
     jbyte value
     ) {
     *((jbyte *)address) = value;
}
```

This results in a much less efficient, as this solution implies a function call (and then costly operations). This even gets worse when accessing physical data which have a size of one bit as programs will often use masks to access the bit in one logical operation, such as the following:

```
REG_FOO |= 0x04; // sets bit 3 to 1
```

The resulting Java code would look like:

```
byte temp;
temp = REG_FOO.read();
temp |= 0x04;
REG_FOO.write(temp);
```

It is possible to improve this by creating whole set of accessors/modifiers as follows:

```
native static void or_mask(int address, byte mask);

// which will perform the following C code:

*((jbyte *)address) |= mask;
```

But the result is not convenient.

Note that [Real-Time Core Specification] addressed these issues by defining some specific APIs to allow I/Os in Java. This brings several advantages :

- The Java APIs are "safer" than raw C code. Some checking is performed when the programmer "opens" the IOPort to make sure the requested range of I/O addresses is accessible to the programmer. This prevents errant pointer access to arbitrary memory locations.
- The Java APIs have higher performance in that (i) these avoid the overhead of JNI method calls, (ii) the compiler will typically in-line their implementation, and (iii) the compiler will typically optimize the code of the surrounding context to make most efficient use of registers, to make use of common

subexpressions, and to eliminate redundant copying of information. Java compilers can do none of this with JNI calls because the compiler is unable to analyze the content of the JNI functions. It must assume worst-case interference between Java and C components.

It is believed that an appropriate implementation of the real-time core IO port API would perform as fast as the C code shown.

4.1.3 Recommendations

4.1.3.1 Development environment

It is recommended that

- embedded static systems change the semantics of the root classes
- standard library is not used
- a preprocessor be available
- debugging interface with suitable features be available

4.1.3.2 System Programming

Extensions at the level of the programming language or features at the level of the compiler should be provided to support constants, macros or in-lining, unsigned types, bit management, interrupt management and I/O management.

4.2 Real-Time Support

4.2.1 Principles

4.2.1.1 Multithreading in Java

Definition

The definition of a thread in Java is the same as in RTOS. A thread is a sequence of code which executes concurrently to others. Threads can possibly share underlying system resources such as files, as well as accessing other objects declared within the same program. Every program consists of at least one thread - the one that runs the main method of the class provided as a startup argument to the Java virtual machine ("JVM"). Other internal background threads may also be started during JVM initialization. The number and nature of such threads vary across JVM implementations. However, all user-level threads are explicitly declared and started from the main thread, or from any other threads that they in turn create.

The standard way to create a thread from a user program consists in writing classes that extend the class called `Thread`, and to implement a `run()` method which represents the thread as showed below :

```
class AppThread1 extends Thread {
     public void run() {
           // code to be executed in a separate thread
     }
}
```

The fact that thread entities are directly managed by the language leads to better portability, because of the independence to the underlying RTOS framework that is used. In contrast using C in general implies the direct use of the underlying RTOS services. Therefore portability would depend on the RTOS API status (in particular using OSEK-VDX API would ensure portability of automotive applications).

Threads are assigned a priority, but here is what the Java tutorial states concerning their use:

"At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use priority only to affect scheduling policy for efficiency purposes. Do not rely on thread priority for algorithm correctness."

The consequence is that standard Java does not support real-time systems.

4.2.1.2 Synchronization

Synchronization is needed when two concurrent entities of more run concurrently access shared resources, in particular shared variables. Such entities could be software threads or hardware interrupt routines.

The following example shows how a shared resource can be accessed concurrently using the `synchronized` keyword.

```
class SharedData {
     synchronized void Access() {
       ...
     }
}
```

Synchronization is not a trivial issue. Consider the following example, taken from [Lea 96-99]

```
final class SetCheck {
  private int  a = 0;
  private long b = 0;

  void set() {
    a =  1;
    b = -1;
  }

  boolean check() {
    return ((b ==  0) ||
            (b == -1 && a == 1));
  }
}
```

In the case where there is only one thread in the program, executing the method `check` will always return true :

- either b is equal to zero
- or b equal -1 and a equal 1.

In the case several threads accessing the methods, there could be potential problems. For example the compiler may rearrange the order of the statements, so b may be assigned before a. This is allowed and would not change the semantics of the program in the single thread case. The following could then happen

```
Thread 1
     a = 0
     b = 0

     call set
           b = -1 (compiler rearranged the code)

Thread 1 is preempted and Thread 2 is scheduled
     Call check
           At this point b == -1, a == 0 so check return false
```

There are many cases where the compiler, the memory system, the processor behave in a non expected way that could cause the example to have an erroneous behavior if the `synchronized` keyword is not used. For instance :

- The memory system (as governed by cache control units) may rearrange the order in which writes are committed to memory cells corresponding to the variables. These writes may overlap with other computations and memory actions.
- The compiler, processor, and/or memory system may interleave the machine-level effects of the two statements. For example on a 32-bit machine, the high-order word of b may be written first, followed by the write to a, followed by the write to the low-order word of b.

- The compiler, processor, and/or memory system may cause the memory cells representing the variables not to be updated until sometime after (if ever) a subsequent check is called, but instead to maintain the corresponding values (for example in CPU registers) in such a way that the code still has the intended effect.

4.2.2 Intended Use in Embedded Systems

4.2.2.1 Response Time Predictability

Real-time systems must meet specified deadlines for the operations they are running in response to external events. In order to achieve this, each of the services which are observable to the user are assigned a maximum response time. For example: when designing the emergency brake of a train, one will specify the maximum time between when the handle is pulled and the moment speed decreases to say 100 milliseconds.

The consequence is that each individual mechanism of the system must in turn execute within a predictable time. This applies to services performed by the underlying RTOS. This applies to the activation of interrupt routine (i.e. interrupt latency). This also applies to mechanisms which have a far reaching impact on the overall behavior such as the scheduling policy or synchronization mechanisms. There are two specific mechanisms in Java which have equally far reaching impact, garbage collection and dynamic inheritance.

Garbage Collection

Most JVMs maintain a heap to store all objects created by an executing Java program. Objects are created through Java `new` operator, which causes memory to be allocated for them in the heap.

Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. The goal of this feature is for the programmer to get rid of the responsibility of tracking all objects and decide for each of them when they should be freed. The term “garbage collection” means that objects that are no longer needed by an application are “garbage” and should be thrown away. It is a break from the usual C programmer’s habit to use `malloc`/`free` services.

A garbage collector is a component integrated into a JVM. It is responsible for:

- tracking all objects that are no longer referenced by the application: that means they cannot be used anymore
- making available the heap space that was used for storing these no longer referenced objects

It should be noticed that the JVM specification does not give detail on the garbage collector except that it must be present in a normal implementation. Consequently, several garbage collector algorithms exist on the market. They are more or less efficient depending on the application type. A desktop JVM will require overall time performance, avoiding memory leaks, even if sometimes the user will have to wait seconds before the garbage collection ends

up, whereas a real-time application will require a more predictable algorithm that in particular can be interrupted when needed without corrupting the heap.

Garbage collection is a crucial issue for real-time systems. It is necessary to guarantee that the garbage collection mechanism can run without provoking unpredictable overhead. A typical approach consists in running the garbage collection as a low priority task so that it can be preempted when a high priority task has to run. But this is not sufficient as the high-priority task may be precisely in the process of allocating memory that is not available.

Lots of research have been carried out to cover for such cases. But to our knowledge only a few products offer real-time garbage collection (Perc from Newmonics (see www.newmonics.com) or Jamaica from Aicas (see www.aicas.com)). We actually have no information on return of experience from applications based on such garbage collection schemes.

Dynamic Inheritance

Because of the object orientation of Java, a mechanism for virtual method call, i.e. invoking the right method in the inheritance scheme is constantly activated. Here is an example.

```
class A {
    void foo() {
    }
}

class B extends A {

    void foo() {
    }
}

A o1 = new A();
B o2 = new B();
O1 = O2;
O1.foo(); // which method is called?
```

Real-time systems expect O(1) algorithms for inheritance schemes.

Remarks from Kelvin Nilsen : Most implementations he is aware of implement the inheritance scheme in O(1). On the other hand, interface invocation is a little more complicated, but that also can be done in O(1) time (NewMonics has such an implementation in PERC).

4.2.2.2 Synchronisation

The Java Memory Model is not Suitable

The Java Memory Model (Chapter 17 of the [Java Language Specification]) defines an abstract relation between threads and main memory. Every thread is defined to have a working

memory (an abstraction of caches and registers) in which to store values. The model guarantees a few properties surrounding the interactions of instruction sequences corresponding to methods and memory cells corresponding to fields. Most rules are phrased in terms of when values must be transferred between the main memory and per-thread working memory. The rules address three intertwined issues:

- Atomicity.
  - Which instructions must have indivisible effects. For purposes of the model, these rules need to be stated only for simple reads and writes of memory cells representing fields - instance and static variables, also including array elements, but not including local variables inside methods.
- Visibility.
  - Under what conditions the effects of one thread are visible to another. The effects of interest here are writes to fields, as seen via reads of those fields.
- Ordering.
  - Under what conditions the effects of operations can appear out of order to any given thread. The main ordering issues surround reads and writes associated with sequences of assignment statements.

Regarding synchronization, the solution derived from the Java Memory Model is to use consistently the `synchronized` keyword, which aims to simplify the characterisation of all the properties above: all changes made in one `synchronized` method or block are atomic and visible with respect to other `synchronized` methods and blocks employing the same lock, and processing of `synchronized` methods or blocks within any given thread is in program-specified order.

The problem is that this model, when applied strictly, can lead to serious speed inefficiency because threads are constantly calling mutual exclusion mechanism.

William Pugh (Univ. of Maryland) [Pugh] raised problems concerning the Java Memory Model (JMM) as described in [Java Language Specification]. Those problems are mainly related to an ambiguous presentation of the problem of ordering described previously. He suggests to describe a new Java Memory Model with more explicit and consistent rules that both allow legitimate aggressive optimization and prevent programmers from writing unsafe programs too easily. This proposition contains, among other definitions, an extended semantics of `synchronized` and `volatile` keywords.

This new model will not be discussed further in this document.

Typical Synchronization Practices in Embedded Systems

Embedded systems typically use various synchronization mechanism. This section describes them. OSEK/OS service names are used but equivalent services can be found in other RTOS.

The following terminology will be used :

- a distinction is made between implicit versus explicit synchronization. Implicit

synchronization happens when an underlying non visible synchronization mechanism is used as a result of the use of a programming language construct.

- A *memory barrier* indicates an instruction for which all the instructions situated before in the code will be executed before, and all the instructions situated after in the code will be executed after.

Here are the possible approaches

- Explicit synchronization : Locks on sections of code

  Two services, `GetResource` and `ReleaseResource` are used to respectively lock and unlock a section. They allow the explicit synchronization of blocks of instructions. Sections that are locked on the same resource cannot be running simultaneously, the first task that gets the resource will be guaranteed exclusivity of this until the resource is released. This technique is used in most conventional systems based on preemptive multitasking (i.e. all tasks are preemptible and schedulable at any time).

- Explicit synchronization: Locks on sections of code involving interrupt routines

  Two services, `DisableInterrupt` and `EnableInterrupt` are used. They are when a thread and interrupt routines need to access critical sections concurrently

- Implicit synchronization: Read/Write atomic data

  Certain types of actions can be considered as "atomic", following the JMM:

  - access to `byte`, `short`, `int`, and reference types.
  - access to variables declared with the `volatile` modifier.

  This means that reading or writing one of these variables is one single indivisible action. When a thread writes a value in such a variable, a concurrent thread reading this variable cannot read an intermediate value of this variable. The value can only, depending on the timing, be the old value (before the write) or the new value (after the write).

  This feature is a very limited means of synchronization (only for a single action on a simple variable) but is commonly used by real-time applications programmers in C. It is not an possible approach if the JMM is not changed as no guarantee is given regarding the order in which data is accessed.

- Implicit synchronization: Non-preemptive programming

  Many embedded systems programmers have been using non-preemptive operating systems for years as a basis for writing multithread applications with a very small footprint. In a non-preemptive thread, a synchronized access to two related variables would be something like :

```
foo() {
    x = 65;
    y = 0;
    Schedule();
    x = 60;
    y = 5;
    Schedule();
}
```

The function above modifies twice the value of (x,y), but we can be sure that the execution of this function will never let the possibility of another thread to see another value than (65,0) or (60,5). Because this other thread can be executed only when scheduling has been allowed by this thread.

It is typical that when non preemptive scheduling is used for synchronization then the `GetResource` / `LeaveResource` services are not necessary.

- Adhoc synchronisation

In this approach, application use an explicit mechanism and an associated access discipline.

A well-known example of these techniques is the use of events or flags for synchronization. Consider two tasks T1 and T2 accessing a shared resource (memory):

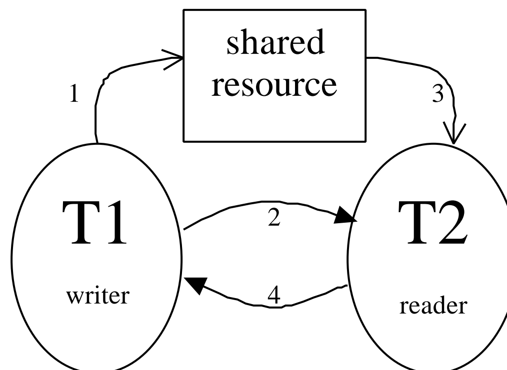


Figure 3: Resource sharing

The natural implementation of this scheme is the following:

```
T1() {
    write_operations();
    SendEvent(T2,AccessPermission);
    other_operations();
}
```

```
T2() {
    int i = 1;
    WaitEvent(AccessPermission);
    read_operations();
    SendEvent(T1,AccessPermission);
    other_works();
}
```

Synchronization Practices Applicable in Java

- Explicit synchronization: Locks on sections of code

  Locks on sections of code is achievable through the `synchronized` construct in Java. When an RTOS is used, the `GetResource` and `ReleaseResource` series are used to implement the semantics of `synchronized.` This means that the Java run-time must make the right call to the RTOS services to reserve a lock resource.

  As a matter of fact, [Java Language specification] does not provide suitable semantics for this type of implementation. One of the reasons is that the `synchronized` semantics is based on a not suitable thread model which, for example, does not provide a valid priority management: no warranty can be made whether a system will block or not, no feature like priority inversion can be provided. As a result a different specification of the semantics of the `synchronized` keyword is required. This was achieved in [Real-Time Core Specification].

- Explicit synchronization: Locks on sections of code involving interrupt routines

  The very notion of interrupts does not exist in the Java language. Therefore some mechanisms must be made available :

  - The approach to have Java threads calling the `DisableInterrupt` and `EnableInterrupt` services surprisingly does not work. This is because Java and its underlying JMM does not guarantee that a section of code located between two calls (e.g. `DisableInterrupt` and `EnableInterrupt`) will be executed in between the two calls. As a matter of fact a Java compiler is allowed to reorder the code and execute portions of code that are placed after `DisableInterrupt` before the call to this service, with the obvious consequence that this call does not protect the desired actions anymore. A solution to this issue is to find a way to specify to the compiler that both services are barriers for optimizations.
  - Another possibility is to reuse the `synchronized` construct which would then be translated into suitable calls to `DisableInterrupt` and `EnableInterrupt`. But then the construct is used for two types of locks (based on `GetResource / LeaveResource` and `DisableInterrupt / EnableInterrupt` respectively). This necessitates a mechanism to specify to the compiler/run-time which service it should use.

- Implicit synchronization: Read/Write atomic data

As stated this approach is not possible unless the JMM is changed so that possible reordering of code cannot be performed by the compiler, or unless compilers not taking advantage of these reordering capabilities are used.

- Implicit synchronization: Non-preemptive programming

  Similarly to direct calls to lock services (e.g. `GetResource`, `DisableInterrupt`) the direct use to a `Schedule` service to achieve synchronization is not possible with the current JMM. Again this means that the compiler would have to be informed that `Schedule` is a barrier.

  Another approach would consist in using the `synchronized` or `volatile` construct. Again if such constructs are used in different context with different lock services, then a mechanism specify to the compiler/run-time which service it should use would be necessary.

- Adhoc synchronization

  Such mechanisms are not possible again because of the JMM. The problem of possible reordering of instructions could change the intended behavior. If we consider the example above :

  - In T1, the `other_operations()` can be executed before `SendEvent()`, but the `write_operations()` must not be executed after `SendEvent()`.
  - In T2, the `i=1` instruction can be executed after `WaitEvent()` whereas `read_operations()` must not be executed before `WaitEvent()`.

  This could be solved if there is a possibility to specify to the compiler that the `WaitEvent` and `SendEvent` services are memory barriers.

4.2.3 Recommendations

4.2.3.1 Predictability

Garbage Collection

It is recommended not to use garbage collection until some experience has been accumulated. More analysis is provided in 4.5.

Dynamic Inheritance

This issue is not specific to Java. It is general to the use of object oriented language for real-time systems. We therefore recommend

- to follow guidelines such as to limit deep hierarchies
- to validate method search algorithms
- to follow return from experience in the use of object oriented languages in real-time systems.

4.2.3.2 Synchronisation

It is recommended to reassess the Java Memory Model first and adopt a more adapted model.

Once a suitable model is available then the following is suggested :

- Use the `synchronized` construct to provide locking sections between threads
- Use the same construct to provide locking sections involving interrupt routines
- Guarantee atomic updates of some basic types.

4.3 Support of Exception

4.3.1 Principles

4.3.1.1 Introduction

Exceptions are widely used programming language entities. They allow programmers to deal with unexpected situations in general. Exceptions as in Java are very similar to those defined in other programming languages such as Ada or C++.

In Java, three main keywords are used :

- `try` is used to define the block where an exception can occur : within the block defined by the `try` keyword, and within any procedure called from the block (nested calls).
- `catch` is used to define portion of code associated with the block called *exception handlers*
- `throws` is used to signal the occurrence of exceptions.

The example below shows a block which calls a method `C.MethodX`. `C.MethodX` contains code which raises exception `ExceptionX`. This causes the execution flow to branch to the catch clause.

```
try {
     C.MethodX();
}
catch(ExceptionX e) { … }
finally { … }

void MethodX throws ExceptionX {
     …
     throw new ExceptionX();
     …
}
```

As shown in the example, throwing an exception often includes the creation of a Java object, instance of a subclass of `java.lang.Throwable`. This object is used to contain information on the exception, describing for instance the cause of the exception.

4.3.1.2 Description

To simplify the further discussion, some technical terms are defined:

A statement (a block, a procedure) the execution of which can result in an exception being thrown is called a *fragile statement* (block, procedure).

If the execution of a fragile statement (block, procedure) ends in an exception being thrown, the statement (block, procedure) is said to complete *abruptly*. Abrupt completion of a statement (block, procedure) causes an immediate transfer of control flow to the appropriate exception handler.

In order to manage the semantics of Java exceptions during the execution of a Java program, an exception context is maintained during the execution. This context defines, at any point *p* in the program, for each exception *e*, the point *p'* to which control flow is transferred if execution of *p* completes abruptly with an exception of class *e* (see Figure 4).

Candidates for such points *p'* are all catch clauses of the Java program. The actual *p'* for a given *p* and *e* is determined at run time as follows: It is the catch clause of the most recently entered try block (which has not yet been left) which can handle an exception of a type *e'* which is assignment compatible to the type of *e*. If a try block contains multiple catch clauses that could handle exceptions the types of which are assignment compatible to that of *e*, the first catch clause in source code order is chosen.

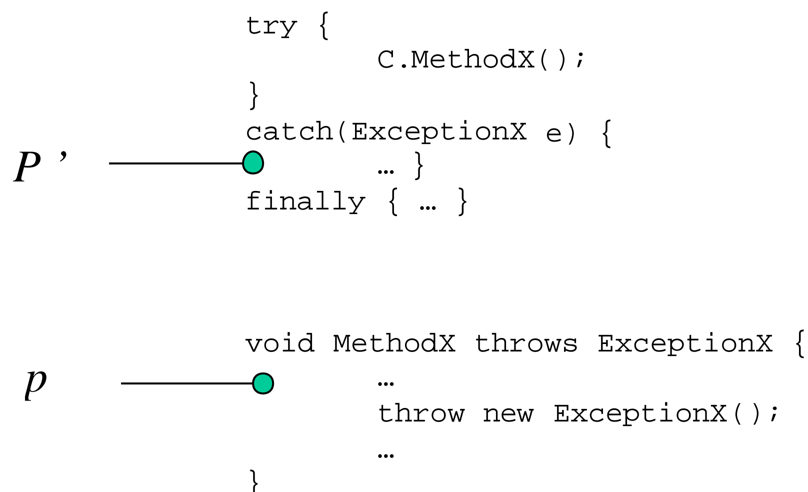


Figure 4. Control Flow Transfer in Java

Note that even the main method has default exception handlers. They are used to catch any exception for which the programmer has not provided handlers. The default exception handlers print a backtrace and then terminate execution of the program.

The Java programming language divides reasons for which exceptions can be thrown into three categories:

- Synchronous, checked exceptions such as indexing an array with an out-of-bounds index.
- Unchecked, asynchronous exceptions, such as the delivery of a stop() signal to a thread.
- The use of a throw statement.

Nested exceptions are not supported in the Java language. At any given time, a Java thread can be handling at most one exception. here is no possibility to retain a current exception through the process of throwing and eventually handling a new exception. In particular, throwing an exception e' in an exception handler (or a finally clause) which was entered as a consequence of an exception e having been thrown previously relieves the thread from taking any action that would have been required by exception e.

There are two principal ways in which the exception mechanism can be invoked:

- an exception is generated explicitly through the use of the Java throw statement,
- by execution an operation (such as a division) under conditions (such as a zero divisor) for which the Java language specification requires an exception (such as a `DivisionByZeroException`) to be raised.

In either case, control flow must be redirected so that - eventually - the appropriate exception handler is executed. Apart from detecting the presence of an exception (object), there are two specific tasks that must be fulfilled :

- the possible intra-procedural control flow that is involved if a fragile method completes abruptly.

  If a statement completes abruptly with an exception for which there is an appropriate catch clause in the current method, then the code contained in this clause is executed and the exception is handled

- The choice of the appropriate exception handler at runtime.

  If a statement completes abruptly with an exception for which there is no appropriate catch clause in the current method, then the method completes abruptly and returns to the caller. The process then continues recursively, i.e. a catch clause is searched at the caller level.

4.3.1.3 Run-Time Implementation Approaches

In order to support exception, specific run-time mechanisms must be made available. We describe below three typical approaches.

Predefined Branch Table Approach

In this approach, a table is consulted after an exceptional condition has been encountered. This table is indexed by a exception identification. It contains the address of the exception handler to call. This approach is widely used in embedded systems because it is mimicking the hardware interrupt approach. The code below is a typical C code.

```
#define EXCEPTIONFOO 3;
BranchTable[EXCEPTIONFOO] = &ExceptionHandlerForFoo
…

RaiseException(EXCEPTIONFOO);
```

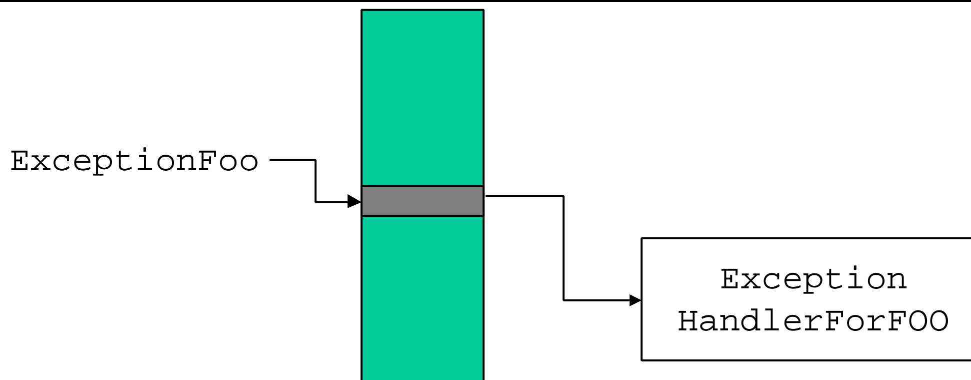


Figure 5. Predefined Branch Table

This approach is not appropriate to support Java because there is no way to compute statically the address of exception handlers in Java, unless some drastic restrictions are made to ensure programming practices (such as catching the exception at the very end of the block where it is thrown, for example).

Explicit Control Flow

In this approach, the control flows explicitly follows the nested calls flow:

- With the method *A* where the exception occurred, the mechanism branches directly to the exception handler if there is one.
- If there is not exception handler, then the mechanism executes a return. An additional return value containing the exception object is transmitted. Its presence means that the caller *B* of the method *A* has to handle the exception. This means searching whether a handler is available (and then branching to it) or else returning the caller of *B*. An improvement of this mechanism is to place the return value in a variable. This is possible because there is only one exception at a time in a Java thread.

If we assume the following example where *C* calls *B* which in turn calls *A*

```
void A throws ExceptionFoo {
     …
     throw new ExceptionFoo();
     …
}

void B throws ExceptionFoo {
     …
     A();
     …
}

void C {
     try {
            B();
     }
     catch(ExceptionFoo e) {
     … }
…
}
```

The real executed code looks like this

```
void MethodA {
     …
     raise exception as follows
     ExceptionInfo.Reason = ExceptionFoo
     ExceptionInfo.Status = Occurred
     return
     …
}

MethodB {
     …
     Call Method A ;
     If Exception.Status equal Occurred then
           Return
     …
}

void MethodC {
     /* try { */
           call MethodB;
     If Exception.Status equal Occurred then
           goto ExceptionArea;

goto AfterExceptionArea;
     /* catch(ExceptionFoo e) */ {
ExceptionArea :
     if ExceptionFoo = Exception.Reason then
           code of ExceptionFoo handler
           goto AfterExceptionArea;
     else
           return
…
AfterExceptionArea :
}
```

This approach incurs the following overhead :

- searching through exception handlers in the current local exception context,
- testing the global exception variable after any point a fragile method has been executed.

Stacked Exception Contexts

In this approach, an explicit searchable representation of the exception context is maintained, and updated at any point in the execution where the dynamic context may change. Upon entering the exception handling mechanism, the explicit context structure is traversed to find an entry which represents the code of the exception handler.

Taking advantage of the fact that new exception handlers can only be introduced with try blocks, which are either properly nested or appear in linear sequence, the context structure can be organized as a stack; its entries are ordered lists which point to exception handlers, and of course identify the exception that is to be handled by the provided handlers. Each time a try

block is entered, a new entry is pushed onto the stack; this entry contains the handlers provided in the catch clauses of this try block in the order in which they appear in the source code. Upon leaving a try block, the entry that has been pushed onto the stack on behalf of this block is popped from the stack. If a try block doesn't provide any handlers, the stack remains unchanged.

Once an exception is been thrown, the search for the appropriate handler proceeds as follows: First, the handler list contained in the top of the stack is searched; if a handler is found, it is executed. Since this means that the try block has been left (to be precise, it has completed abruptly), this top entry is popped from the stack prior to actual execution of the handler code. If no applicable exception handler is found in the handler list of the top entry, the top entry is popped as before (since control leaves the try block again), and the new top entry is searched as before.

For instance, the following code

```
void MethodA {
     try {
           try {
                 throw new ExceptionFoo();
…
           }
           catch(ExceptionFoo e) { // address2
           … }

     }
     catch(ExceptionFoo e) { // address1
     … }
     catch(ExceptionBar e) {
     … }
…
}
```

Would imply the following execution

```
void MethodA {
     enter first try
     push(address1)
           enter second try
           push(address2)


           Raise exception
           address = pop()
           Branch to address
…
           pop
           leave second try
     pop
     leave first try
…
}
```

4.3.2 Use In Embedded Systems

4.3.2.1 Software Engineering

It is generally agreed that exceptions are suitable to prevent the transformation of a clean program structure into a spaghetti-like structure when error management has to be included. See [Sun exception] for a convincing explanation.

However, the use of exceptions in embedded systems has raised significant controversy. It has been widely argued [Hoare81] that most exception approaches are unsafe for embedded systems :

- When the used programming language uses a dynamic search for exception handlers, which is the case of Java, Ada, and C++. It is argued that it is difficult to validate that the right exception handler will be used[4]. Consequently it is difficult to validate a program using exceptions.
- When a non appropriate default handler is used for unexpected exceptions. For instance the occurrence of an exception which is by design supposed not too happen (i.e. a bug) might get to the default handler which would stop the running program and lead to catastrophic failures.

4.3.2.2 Real-Time Behavior

The handling of dynamic exceptions involve

- systematic overhead. In the explicit control flow approach, overhead is systematic upon returning from a method call. In the stacked exception context approach, overhead is

[4] One could argue that branching to an exception handler is even worse that a goto statement (an unanimously criticized programming construct). In the goto statement, the execution flow of a program is abruptly changed to a statically defined location. In the branching to an exception handler, the execution flow of a program is abruptly changed to a dynamically defined location.

systematic upon entering and leaving a try block[5],

- exception occurrence overhead to dynamically search for the exception handler. In the explicit control flow approach, the overhead depends on the number of nested methods calls. In the stacked exception context approach, the overhead depends on the number of pushed exception handlers.

Systematic overhead is likely not to exceed a multiplying constant *C*. *C* is acceptable as long as the overall systematic overhead due to Java does exceed a given threshold.

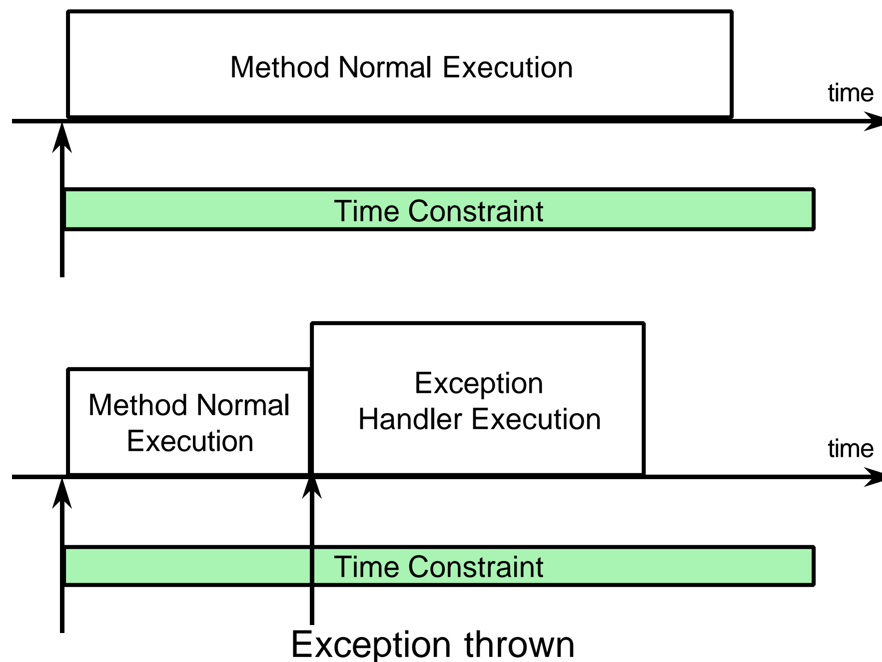


Figure 6 Timing Constraint at Method Level

Exception overhead occurrence depends on whether a real-time constraint is associated with the application "process" in which the exception is raised or with the exception handler itself. In the first case, the overhead is systematic and therefore predictable (see Figure 6). In the second case, there is a predictability issue related to the dynamic search (see Figure 7). The second case is therefore not an acceptable behavior for embedded systems.

---

[5] Note that this would promote Java programming style where a try block containing a loop should be preferred to a loop containing a try block

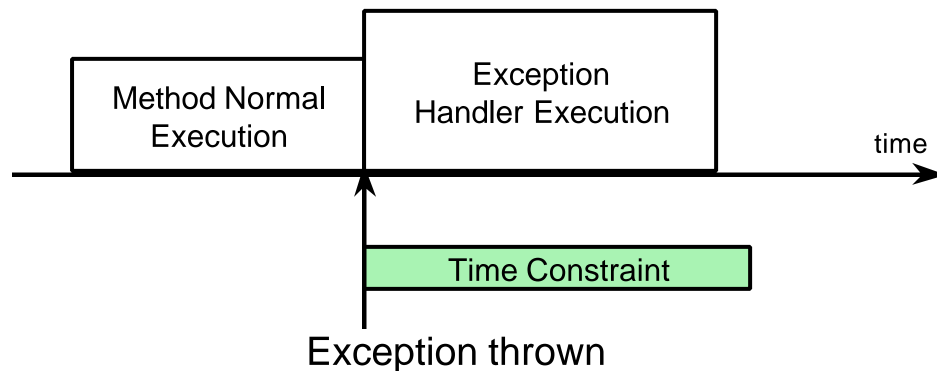


Figure 7 Timing Constraint at Handler Level

4.3.2.3 Memory Needs

Exceptions in Java are objects which are usually instantiated each time an exception occurs. The rationale behind this approach is to allow for the provision of specific information about the exception. For instance, the exception object could contain data to be made available.

This means that each time an exception is raised, then some memory resource is allocated. This is not an expected behavior in embedded systems, especially in static systems where all resources are predetermined.

Fortunately Java allows applications program to throw exception objects which have been pre-instantiated.

An important issue is that the standard behavior of predefined system exceptions (e.g. `ArithmeticException`) is to instantiate dynamically exceptions. This is an issue for embedded systems.

4.3.3 Recommendations

4.3.3.1 Software Engineering

The following recommendations can be made for embedded systems :

- Because exceptions can be used in a unstructured way, i.e. an unsafe way, apply software programming rules, such as making sure exception handling code appear close to the fragile statement it handles exception for.
- Make sure that a default handler is associated with each runtime error (in order to avoid a default handler which stops the program).

4.3.3.2 Real-Time Behavior

The following recommendations can be made for embedded systems :

- Exception handling carries a systematic overhead which should be taken into account.

- If a timing constraint must be associated with an exception handler, then make sure the dynamic search overhead for a handler is predictable, possibly by enforcing software programming rules (for example, verify the call nesting level as it is already typically done in Embedded Systems to avoid stack overflow).

4.3.3.3 Memory Needs

The following recommendations can be made for embedded systems :

- It is highly recommended not to instantiate a user-defined exception in the throw statement[6]. Rather, create it at the very beginning of the program and throw it when needed (see example below). This could even allow them to be ROMized.

```
throw new FooException()
```

is replaced by

```
// in the init part
FooException MyInstanceOfFooException = new FooException();

// in the method code
throw MyInstanceOfFooException;
```

- Likewise, the same recommendation applies to runtime exceptions, i.e. make sure the Java runtime does not create objects when an exception is thrown upon a runtime error.

4.4 Initialisation

4.4.1 Principles

4.4.1.1 Introduction

This section analyzes the initialization characteristics of Java software programs and of embedded systems software. It then provides recommendations concerning the initialization of Java components for embedded systems. In order to clarify discussion, it is convenient to describe the execution of a software component with the following phases:

- Download phase : the software is placed where it will be executed.
- Startup phase : initialization is performed
- Active phase: services are available and can be executed on-demand.

[6] Note that this "breaks" tradition in that traditional Java exceptions carry a stack backtrace to identify where the exception was thrown from.

4.4.1.2 Initialization in Java

A Java component consists of a set of classes. Its initialization must therefore allow for the proper initialization of these classes. Two parts have to be considered :

- A static part consisting of class fields and methods. They are similar to global variables and functions in non-object oriented languages such as C.
- A dynamic part consisting of objects, or dynamically created entities which include their own fields and methods. Objects have no counterpart in non object-oriented programming languages such as C.

The initialization of a Java component (and in general of object-oriented programs) must take into account the initialization of the static part as well as the dynamic part. This is done as follows in Java (as per [Java Language Specification]) :

- Downloading takes place at the class level at anytime. Note that downloading may imply more than just copying code and data into memory. It could also involve such as dynamic linking, or even de-serialization of classes (i.e. reconstructing a class from a stream)
- Classes are initialized at first use
- Objects are created and initialized at any time.

Therefore the initialization of a Java software component follow the download, startup and active phases in a non classical manner, as some parts may already be in the active phase while others are just being loaded or initialized.

The following sections explains in further detail class and object initialization as specified for a classical Java environment.

Class Initialization at First Use

Class initialization is highly influenced by the underlying capability that classes can be downloaded at anytime. It involves

- the possible recursive download and initialization of classes.
- the initialization of internal run-time data for each class
- the execution and initialization of user programmed entities associated with classes such as class static initializers (1), initializers for static fields (2). This scheme is described as follows in [Java Language Specification] :

  "Initialization of a class consists of executing its static initializers (1) and the initializers for static fields (2) declared in the class.

The example below shows both types of initialization.

```
class Foo {
     static int c = 5; // (2)initializer for static field
     static int d;
     static {          // (1) class static initializer
           d = 6;
           }
     }
```

The events which can cause a class T to be used for the first time are the following :

- (1) T is a class and an instance of T is created.
- (2) T is a class and a static method declared by T is invoked.
- (3) A static field declared by T is assigned.
- (4) A static field declared by T is used and the reference to the field is not a compile-time constant
- (5) T is the superclass of a class U which needs to be initialized

The following example, shows the five cases of initialization.

```
class A {}
class B {
     static void foo();
}
class C {
     static int bar;
}
class D {
     static double r = Math.random();
}
class E {}
class F extends E{}
class App {
     void main() {
           A a = new A();          //(1) class A is initialized
           B.foo();                //(2) class B is initialized
           C.bar = 42;             //(3) class C is initialized
           if (D.r> 0.5) {         //(4) class D is initialized
                 F f = new F();    //(5) class E is initialized and then
                                   //    class F is initialized
                                   //  note the E and F
                                   //  may never be initialized if they
                                   //  not used elsewhere
           }
     }
}
```

Interface Initialization at First Use

Classes could implement interfaces. They also follow a philosophy of initialization at first use. Apart from run-time internal initialization code, user programmed entities also need to be initialized. [Java Language Specification] specifies the following :

"Initialisation of an interface consists of executing the initializers for fields declared in the interface".

This shown in the example below

```
interface B {
     int b = 4;        // initializer of an interface field
     }

class Foo implements B {
     …
     }
```

The events which can cause an interface to be used for the first time are the following :

- (1) A static field declared by T is assigned.
- (2) A static field declared by T is used and the reference to the field is not a compile-time constant

Note that interfaces implemented by classes are not initialized at class first use. Similarly, superinterfaces of an interface need not be initialized before the interface is initialized.

The following example, shows the two cases of initialization.

```
interface C {
     static int bar;
}
interface D {
     static double r = Math.random();
}
class E {}
class F extends E{}
class App implements C, D {
     void main() {
            C.bar = 42;             // (1) interface C is initialised
            if (D.r> 0.5) {         // (2) interface D be initialised
            }
     }
}
```

Object Initialization

Object initialization involves

- some internal run-time overhead. At the beginning of an object's life, the Java virtual

machine (JVM) allocates enough memory on the heap to store the object's instance variables. If there is no explicit initialization of instance variables, then some default values are assigned : zero value for numeric types, null for a reference and false for a Boolean. See the example below. Note that local variables are not given default initial values.

```
class MyClass {
     private int bar; // bar is initialised to zero
     void foo() {
           int a; // a is not initialised
     }
}
```

- The execution of user programmed code to initialize the instance variable. Three mechanisms are available to ensure proper initialization of objects: (1) instance initializers (also called instance initialization blocks), (2) instance variable initializers, and constructors (3). (note that instance initializers and instance variable initializers collectively are called "initializers."). All three mechanisms result in Java code that is executed automatically when an object is created.

  The example below shows an instance initializer.

```
class MyClass {
     private int foo;
     { // This is the instance initialiser block
           foo = 355; // foo is initialised to 355
     }
}
```

The example below shows an instance variable initializer

```
class MyClass {
     private int foo = 355; // foo is initialised to 355
}
```

The example below shows an constructor

```
class MyClass {
     private int foo; // foo should be initialised to 355

     MyClass {
           Foo = 355;
     }
}
```

More detailed rules and examples can be found in [Venners 98].

4.4.1.3 Initialization in C

In comparison, initialization in C is much simpler. C does not propose any programming constructs for initialization except something that can be considered as the equivalent of Java initializers for static fields.

A software component written in C is viewed as a collection of c files. The only entities to initialize are global variables. Two schemes are available :.

- (1) initialization of variables declared as constants. In most implementations, this means that the variables can be located in ROM. If we refer to the download, startup and active phases, such variables are therefore initialized during the download phase. Note that Java does not allow the declaration of constants.
- (2) initialization of other variables. This means that such variables will be located in RAM. Such variables are then initialized during the start-up phase.

Here is an example of the two types of initialization

```
const int var_in_rom = 42;  // (1) initialized at download phase
…
int var_in_ram = 12;        // (2) initialised at start-up phase
…
```

4.4.1.4 Java versus C

This presentation shows that

- Java provides a initialization programming model. C does not.
- Java does not provide to the program an explicit way to control the initialization execution. This is not the case in C precisely for the reason that C has no initialization programming model.
- C allows for the declaration of constant data which can be then be put in ROM. Standard Java does not.

4.4.2 Intended Use in Embedded Systems

4.4.2.1 Practices

Typical Practice for Initialization

Typical initialization in embedded systems written in C is performed during the startup phase in order to guarantee a clear view of the initial state of the system before active phase

- (1) global variables are initialized before execution of any user code. They are initialized at the very beginning of start-up time, before execution of the `main()` function,
- (2) initialization of each software module. Each module provides an initialization function which is called explicitly during start-up time, typically in the `main()` function.

The following example gives an idea of what a system initialisation can be:

```
int Foo = 12;          // (1) initialised at start-up phase before main

main() {               // (2) initialization of each software modules

   Hardware_init();          // for example : sets wanted interrupts
   Drivers_init();           // sets internal data to manage present
                             //  devices
   HMI_init();               // sets structures linked to HMI
   statemachine_init();      // sets state machine to initial state

// initialization is completed, active phase is starting
   while (1)
      statemachine_exec();  // active mode : state machine is running
}
```

Specific Practice for Static Systems

Embedded static systems have further specific requirements which are due to the type of memory footprint they are using (for instance a system on a chip with 2kbytes RAM, 16 Kbytes ROM), and the need for a predictable usage of such resources.

The resulting practice is to have as much as possible data in ROM. This means that whenever possible data that are known to remain constant during the active phase should not be calculated and initialized during the startup phase. This should be done before, i.e. during the downloading phase of data into ROM. This practice was a requirement for instance in the definition of a real-time kernel standard such as OSEK [OSEK], where it is assumed that kernel data such as the tasks descriptors will be partially in ROM.

4.4.2.2 Class Initialization

As it should be clear now, the use of standard Java implies initialization approaches that are not compatible with embedded systems. Some further guidelines and possibly modification of the initialization scheme in Java is therefore necessary :

- Dynamic class loading is not supported. There is no guarantee that a universal downloading mechanism can download a class in time.
- Initialization of classes does not take place at first use in order to avoid non predictable execution time.

The following approach is possible in order to avoid class initialization at first use :

- All classes are initialized during the starting phase at the very beginning of the program before any Java code is executed. This approach makes initialization entirely predictable, but it raises a major issue, initializing classes with the right order. In order to handle that, we can see two techniques :
  - The order in which the classes are initialized is given by the programmer (e.g. through annotations). This is an error prone approach. The programmer could make a mistake and then would have to debug class initialization. Secondly it could lead him to have non portable code that rely on an explicit ordering.

- The order in which the classes are initialized is pre-computed by the compiler and all ambiguities (e.g. circular dependencies) have to be removed by the programmer.

Note that initialization at first instantiation (when an object of class is instantiated (created) it can check if the class is initialized; it will initialize the class if it is not initialized) is not a correct approach :

- static fields of a class could be used (read/write) before an object of this class is created
- even if a class has no static field, this approach is not satisfactory because the instantiation overhead is not predictable from one instantiation to another.

4.4.2.3 Object Initialization

Similarly, in order to gain predictability, it could be desirable that objects are initialized before getting to the active phase; e.g. during the startup phase. We call such objects *pre-initialized or predefined objects.* They are objects which the application can access from the very beginning of the execution.

If class initialization is executed during the startup phase, no further complex mechanism is needed. The embedded system programmer just has to declare the predefined objects during class initialization phase, and tell the compiler that they have to be initialized before other kinds of classes, using any kind of annotation (ex: special class name, implement a compiler-known interface…). This is illustrated in the example below

```
public class PredefinedObjects {

    /* Class Fields that store the Predefined Objects: */
    public static Class1 Object1;
    public static Class2 Object2;

    /* Initialization, creation of the predef objects: */
    static {
    Object1 = new Class1(…);
    Object2 = new Class2(…);
     }
}

// somewhere else in the application:
public class Application {
    public static void main(String[] args) {
       // ....
       Object1.method1(…)
    }
}
```

4.4.2.4 Constants in ROM

A critical issue is to allow the use of constants in ROM. Because standard Java does not support constants, an investigation has to be made on possible extensions :

- Use of constants accessed through native methods. This solution is straightforward and does not require extensions to a standard Java system. On the other had it implies the systematic overhead of method call. In practice this is an acceptable approach when there is not so much ROM data, or when the compiler is able to inline C code.
- Support of objects in ROM, or objects which include constant fields. This approach implies programming language investigation, i.e. how can such objects be declared and how a compiler can verify that an object can be ROMized. Objects in Java have a complex initialization phase. In order to validate that an object can be located in ROM, a complex analysis would have to be performed to verify 1) that all fields are constants, 2) that all field will not be changed (written) and 3) that there is only one object. This kind of analysis is required for arrays in ROM and stackable objects, also.
- Part of the startup code of the application is executed before download phase on the host system. Resulting data are frozen and put in ROM. This approach was not investigated.

Supporting ROMized Objects

The rest of this section explains some of the investigation that was carried out for the support of objects in ROM.

First of all here are a number of considerations that have to be taken into account

- ROMized objects could contain internal fields which cannot be located in ROM. If this is the case, then appropriate mechanism must be included at run-time to access ROM area as well RAM area.
- An algorithm has to be found that allows a compiler to determine for a given object whether it can be ROMized. This would be straightforward in the case of objects of type "array of scalar type". These are arrays that are initialized at the very time they are declared, like in the example below. This would raise issues for more general type of objects such as checking if the initialization value is known at compile-time, checking if the value of an object field remains constant, or determining the instantiation points.

```
// example of arrays of scalar types
class MyClass {
    static int staticfield_array[] = {1,2,3,4,5};
    int instancefield_array[] = {6,7,8,9};
    void foo() {
        int local_array[] = {2,4,6,8};
        ....
    }
....
}
```

In an approach where the programmer is responsible for determining the objects that should be ROMized then some solutions are possible. We describe two of them.

Supporting ROMized Objects through Data Declaration in Separate Files

This solution consists in declaring data in separate files. Let us suppose we want to ROMize some instances of the following class:

```
class Foo {
    final int a;
    final int b;
}
```

The ROMized objects could then be declared in a configuration file like the following:

```
ROM "foo1" OF CLASS "Foo"
{
    a = 7;
    b = 23;
}
```

And a means to have a reference to this object could be the following:

```
Foo foo1 = getROMized("foo1"); /* instead of new Foo() */
```

This approach allows full control of the ROMization by the programmer. But it requires the definition of a romization language and the availability of appropriate tools.

Supporting ROMized Objects through Manual Creation of Objects in C

In this approach, the programmer describes in C the initialization values of a const C structure which directly maps onto a Java object. This necessitate the knowledge of this mapping. For instance the following structure declaration in C could be the counterpart to class Foo above.

```
struct _Foo {
    _Class * _class;
    javaint  a;
    javaint  b;
}
```

Then it is expected that the programmer performs initialization by declaring a C file with the following code

```
const struct Foo foo1 = {
     &Foo_Class,
     (javaint)7,
     (javaint)23
}
```

This approach is straightforward and simple to understand. On the other hand it is highly dependent on the physical layout used by the compiler and may thus render the code non portable.

Supporting Static Final Fields as Constants

In this approach, the only constants that can be placed in ROM, are static final fields, which are initialized with a compile-time constant value, e.g.

```
class foo {
  final static int bar = 5;
}
```

The generated code looks then like:

```
AJACS_inROM int_J foo_bar = 5;
```

A further optimization would still be possible consisting in replacing all reads of this field by the constant itself. The fields do exist in case, that the C code reads these fields:

```
class D {
  int i = foo.bar;
}

  *( object_ptr + offset(i) ) = 5;
```

4.4.3 Recommendations

4.4.3.1 Class Initialization

It is recommended

- to download all classes at compile-time,
- to execute class initialization during the startup phase. This necessitates from the compiler specific analysis capability at the level of initialization expressions for object instantiation, create class dependencies (class c1 must be initialised before class c2) and subsequently generate the correct sequence of class initializations. It is possible that the compiler cannot determine this sequence. It is proposed that in such case the compiler reject the resulting program and issue compiler error messages.

4.4.3.2 Object Initialization

It is recommended to use the predefined object approach for embedded systems which require a static definition of objects.

4.4.3.3 Constants in ROM

The support of constants in ROM may be one of the most important issue to solve. We advocate the use of Static Final Fields as Constants. In the case where full objects have to be put in ROM, then rurther investigation is needed with possibly the need for the Java compiler community to agree on some approaches that would need to be standardized.

4.5 Memory Management

4.5.1 Principles

4.5.1.1 Garbage Collection

See 4.2.2.1.

4.5.1.2 Usage of the new Statement

The [Java Language Specification] does not explicit list all the cases when objects are created. There are two cases :

- instantiation of a user-defined class
  - this is the straightforward case and is a direct consequence of the object-oriented software engineering of the application.
  - it is likely that in a well-defined software module, the programmer will create only the objects he needs.
- creation of intermediate objects for adaptation to an interface
  - this is a non straightforward case which can be typically found out in Java components which interact with already existing libraries.
  - it is often due to a break in the object-oriented scheme. For example, consider the program below which first reads into a array a sequence of characters, and then invokes a printing function with a String parameter

```
char myArray[];
myArray = readArray(); // this method returns a new char[] instance
printingFuntion(new String(myCharArray)) // this method needs a String
```

Memory used for such a temporary use has no real application justification. But it is just handy and it is known that it be reclaimed later by the garbage collector anyway.

4.5.1.3 Implicit Creation of Objects

Programmers using third party packages of Java classes often have no sufficient knowledge of the objects created by these libraries. This even apply for "standard" libraries delivered by Sun Microsystems. For instance, the use of strings and exceptions often lead to the internal creation of objects the programmer is unaware of. This is illustrated in the following example:

```
int i=42;
String s = "myString = " + i;
```

While this portion of code contains no explicit "`new`" statement, it will actually create:

- a `String` object for the `"myString = "` string.
- a `String` object in an internal method (`Integer.toString)` called implicitly to convert 42 in a `String`
- an array of 12 `chars`, used internally in the implementation of `Integer.toString` in JDK1.3
- a `StringBuffer` used for performing the append ("+") operator
- a `String` object for `s`. This is the only "visible" implicit call to `new`.

Except for `s`, all the other objects will never be referenced further by the program, and then could be garbage collected right away.

Exceptions may also lead to uncontrolled object creation, for example, suppose the following code :

```
int myDivide(int p, int q)
{
     int res;
     try {
          res = p/q;
     } catch (ArithmeticException e) {
          res = 0;
     }
     return res;
}
```

Then each call to this function with a zero as a second argument will lead to the creation of an intermediate `ArithmeticException` object (division by zero) that will have no referencing at the time the function returns, and then will remain in memory until it is garbage-collected.

4.5.1.4 Downloading Overhead

The format of Java class files, as defined in the [JVM Specification], does not allow them to be executable in an efficient manner. A JVM implementation will have first to copy almost all the content of these classes into a specific memory area viewed as a temporary disk, and then perform a loading into implementation-defined structures, before being able to execute the code. This is a non negligible memory overhead.

4.5.1.5 General Practice Recommendations

Even in typical Java environments where vast amount of memory is available, the uncontrolled use of services described above could lead to a lack of free memory.

String concatenation using `StringBuffer`

The `String` concatenation feature is a good example of a potential misuse of the standard API which was detected and corrected at the very beginning of the Java language existence. The fact that `String` strings are immutable (i.e. they cannot be modified) increases the number of objects needed, so the user is prompted to use directly the `StringBuffer` class, even if it doesn't sound natural.

According to the recommendations of the Java specification, the previously cited example:

```
int i=42;
String s = "myString = " + i;
```

should be rewritten this way:

```
int i=42;
StringBuffer temp = "myString"
temp.append(i);
String s = temp.toString();
```

which will create

- a `StringBuffer` object for the `"myString = "` string.
- an array of 12 `chars` used in the internal implementation of `Integer.toString` in JDK1.3
- a `String` object in `Integer.toString` called implicitly to convert 42 in a `String`
- a `String` object for `s`.

Note that no new object is needed for the append operation, as `StringBuffer` strings can be expanded (they are not immutable as `String` objects).

Re-use existing objects when possible

Another way to avoid the creation of more objects than needed consists in designing an application in such a way that objects can be re-used several times. Several instances of one class are created.

Assume the following example, where one wishes to draw several points:

```
class Point {
     private int x;
     private int y;
     Point(int x, int y) { this.x=x; this.y=y;}
     void draw();
}

class Application {
     void foo() {
           Point p1 = new Point (42,0);
           p1.draw();
           Point p2 = new Point (195,0);
           p2.draw();
           Point p3 = new Point (84,12);
           p3.draw();
           Point p4 = new Point (126,523);
           p4.draw();
     }
}
```

If the Point class was redesigned to allow modification of its fields, as follows, the code can be written while creating only one Point object:

```
class Point {
     private int x;
     private int y;
     Point() {}
     public void set(int x, int y) { this.x=x; this.y=y;}
     public void draw();
}

class Application {
     void foo() {
           Point p = new Point();
           p.set(42,0);
           p.draw();
           p.set(195,0);
           p.draw();
           p.set(84,12);
           p.draw();
           p.set(126,523);
           p.draw();
     }
}
```

Creation of object pools

The reuse of existing objects described above can be made automatic using a particular pattern called "object pools". These object pools are collections of objects of the same class that can be reused with other values when their content is not needed anymore.

Note that this scheme is only efficient when using large collections of instances of the same class, with both numerous allocations and numerous de-allocations. Else, the normal way of proceeding is often sufficiently efficient. In order to avoid memory leaks, it is also suggested to define an accurate default size for such collections and pre-allocate all the necessary objects.

A implementation of such an object pool can be the following:

```
class myClass { // the kind of objects to gather in a pool
     private int foo;
     private byte bar;
     public void set(int foo, byte bar) {
           this.foo = foo;
           this.bar = bar;
     }
}
class myClassPool { // the pool of myClass objects
     private myClass pool[];
     private boolean valid_obj[];
     private int nb_reusable_obj;
     myClassPool(int nbobj) {
           int i;
           nb_reusable_obj = nbobj;
           pool = new myClass[nbobj];
           valid_obj = new boolean[nbobj];
           for (i=0; i<nbobj; i++) {
                 pool[i] = new myClass();
                 valid_obj[i] = false;
           }
     }
     public myClass createObj(int foo, byte bar) {
           int i;
           for (i=0; i<nb_reusable_obj; i++) {
                 if (valid_obj[i] == false) {
                       // this pool object is not used yet, return it
                       valid_obj[i] = true;
                       pool[i].set(foo,bar);
                       return pool[i];
                 }
           }
           // the pool is full: return an unmanaged instance of myClass
           myClass newObj = new myClass();
           newObj.set(foo, bar);
           return newObj;
     }
     public void releaseObj(myClass obj) {
           int i;
           for (i=0; i<nb_reusable_obj; i++) {
                 if (pool[i] == obj) {
                       // mark the object invalid
                       valid_obj[i] = false;
                       return;
                 }
           }
           // the object is unmanaged: trust the garbage collector
     }
}
```

In this implementation, the programmer will create an object pool of a predefined size (which will never diminish), and if he needs more objects, they will be created using the default JVM mechanism. The example of the previous paragraph could look as follows in such an implementation:

```
class Application {
     void foo() {
          PointPool pool = new PointPool(2);
          Point p1;
          Point p2;
          p1 = pool.create(42,0);
          p2 = pool.create(195,0);
          p1.draw();
          p2.draw();
          pool.releaseObj(p1);
          pool.releaseObj(p2);
          p1 = pool.create(84,12);
          p2 = pool.create(126,523);
          p1.draw();
          p2.draw();
          pool.releaseObj(p1);
          pool.releaseObj(p2);
     }
}
```

The resulting programming practice therefore reverses to a situation where one have to care with the release of objects. If `releaseObj` is not called for a particular object, then the object is lost forever, as the object will never be garbage-collected, since it is referenced in the pool.

4.5.2 Intended Use in Embedded Systems

4.5.2.1 Limited amount of memory

Even though memory cost is constantly decreasing, limitation of memory is still a concern in embedded systems. Prices of consumer electronics devices such as PDAs precisely depend on the amount of memory available.

One of the main issues is the downloading overhead of Java classes in terms of memory. We saw previously that the Java class file format incurs significant memory overhead. Solutions based on the use of a different formats could allow efficient execution if the format allows the downloading phase to avoid intermediate copies from memory to memory. But these formats would have to be standardized in order to guarantee downloading in every platform. This is the case of the ISO standard JEFF file format [JEFF]. Further, the JEFF format is fully reversible in that a JEFF file can be converted into a class file and vice-versa.

4.5.2.2 Static Embedded Systems

Because of the very limited use of RAM memory in static embedded systems (for instance 2kbytes RAM), very stringent limitations would have to be considered :

- In terms of downloading, compile Java classes into native code. This raises platform independence issues, but it allows the execution of code placed on ROM.
- In terms of programming, follow some guidelines such as :

- create most objects at the very beginning of the execution of the application, or even at generation time (see section on initialization)
- avoid the dynamic creation of objects if possible in the rest of the program. Possibly use object pools

- In terms of JVM, use an implementation which
  - Provides a good traceability on how objects are mapped into memory.
  - Support the possibility to allocate temporary objects in the stack (for instance as an extension to the language as proposed by [Real-Time Core Specification].

4.5.2.3 Hard Real-time Systems

See 4.2.2.1 for an analysis of the relevance of using garbage collection in real-time systems.

Another alternative is therefore not not to use a garbage collector. As a result, either a fully static system where all objects are allocated at the very beginning is defined, or a mechanism to de-allocate objects is made available.

One solution for a JVM to provide de-allocation feature without breaking completely the Java philosophy is to organize the memory into banks that group objects and can be de-allocated as a whole by the programmer, for example when a high level treatment is finished. This is exactly the solution provided by the [Real-Time Core specification], for example with the `AllocationContext` service: each piece of code executes within an active `AllocationContext`, which groups all the new objects (created with new), and the programmer has the possibility to free an `AllocationContext` as a whole.

4.5.3 Recommendations

4.5.3.1 Limited Amount of Memory

For systems that are weakly constrained, the main recommendation is to use a Java environment that is able to execute from ROM memory, such as a JVM based on JEFF classes description.

The programmer should also apply all the techniques described in this document and other related book from the Java community in order to use efficiently the JVM., such as memory pools, `StringBuffer` instead of `String`, etc….

4.5.3.2 Static Systems

It is recommended to follow the guidelines listed above in 4.5.2.2 as well of the mechanisms for initialization described in section 4.4.3.

4.5.3.3 Hard Real-Time Systems

It is recommended not to use garbage collection until some wealth of experience has been accumulated. Instead a mechanism based on memory banks is advised.

4.6 Native Interface

4.6.1 Principles

4.6.1.1 Introduction

A Java native interface is needed in two types of applications:

- Support for legacy software code. It can either be legacy code which Java applications have to access or the other way round, i.e. Java code which legacy applications have to access.
- Replacing a Java implementation by a more efficient legacy code implementation. For instance, as interrupts often have the requirement to be treated very quickly, the most time-constrained Interrupt Service Routines can be written in C.

In order to allow the mixing of Java and other programming language code, Sun Microsystems has defined a mechanism called Java Native Interface [JNI]. We first explain the various native interface exchanges in JNI and then discuss possible native interface profiles.

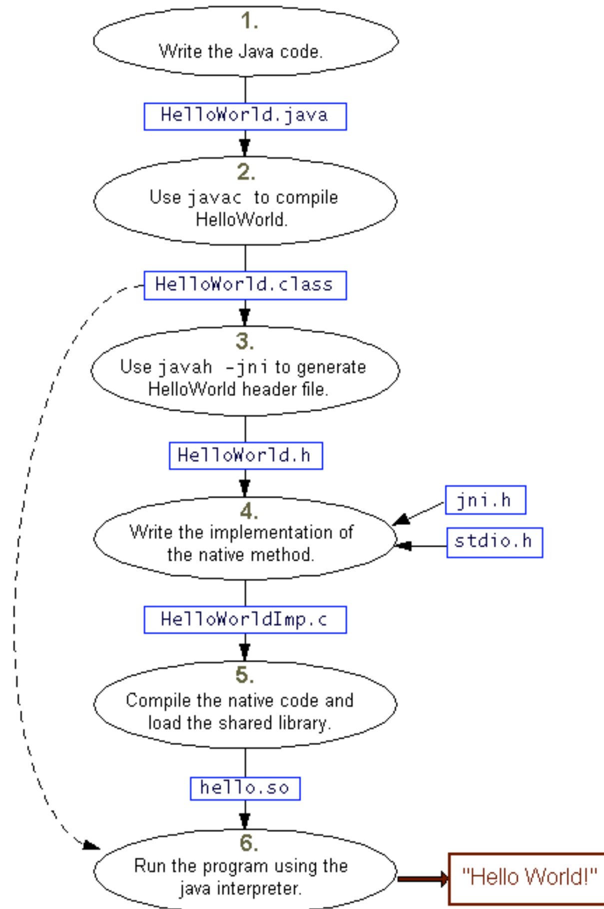
In the rest of the section, we will assume that legacy code is C code.

4.6.1.2 JNI Mechanism to Call C

The most useful interface is probably the function call mechanism. It is typically used when a new program, written in the Java language, reuses legacy C functions written in C. Such C functions are declared as native methods as follows

```
class HelloWorld {
     native void hello();
}
```

The steps for implementing such a function with the usual JNI is split in 6 steps, as the following figure shows (extract from the [JNI tutorial]):



Here is a short description of the six steps :

- First step : some Java code is written to access a method that has been declared as native.
- Second step : the Java code is compiled
- Third step : a Java tool called Javah is used to generate C headers file
- Fourth step : A C function conforming to the generated C header file is written. . The `hello()` method given above then looks like the following

```
JNIEXPORT void JNICALL Java_HelloWorld_hello (JNIEnv *env,
                                              jobject obj) {
// C code for the method
}
```

Note that

- Some specific header convention are used (`JNIEXPORT` and `JNICALL`)
- Some specific parameters are provided, the JVM environment (`env`), along with a reference to the self object (`obj`)[7].

[7] As C components have access to object references they may decide to store in some C variables the reference and use them later, for instance in a subsequent native all. Some precaution must be taken because the validity of object references may change overtime.

- Fifth step : the C function is compiled and loaded
- Six step : the application is executed

4.6.1.3 JNI Mechanism to Call Java

This it typically used when a C program calls a Java method.

Let us assume that the C component seen above wishes to invoke the `foo` class method:

```
class HelloWorld {
     native void hello();
     …
     static int foo(int i) {return i + 1};

}
```

In order to achieve this, three steps are defined :

- get a reference to the class of a given object through a JNI service
- get a reference to the method to call through a JNI service
- request for a method call through a JNI service

The C code used to call this method is the following:

```
JNIEXPORT void JNICALL Java_HelloWorld_hello (JNIEnv *env,
                                               jobject obj) {
// env is a context
// obj is a reference to the self object


     // step 1
    jclass cls = (*env)->GetObjectClass(env, obj);

     // step 2
    jmethodID mid = (*env)->GetStaticMethodID(env, cls, "foo", "(I)I");
    if (mid == 0) { // check identifier validity
        return;
    }

    // step 3
    depth = (*env)->CallStaticIntMethod(env, cls, mid, depth);
    ...
```

Note that step 2 involves a dynamic search on the name of the method (a string 'foo"), as well as on the signature of the method (a string "(I)I"). The signature is used to make the difference between two methods with the same name.

4.6.1.4 JNI Access of Java Data in C components

Once a C function is running, it may wish to access parameters, or Java structures.

The following types of data are defined in Java :

- integral types (`byte`, `int`, `bool`, ...). The possible operations are read and write.
- references to Java objects. The possible operations are creation, read and write a field, method calls.
- arrays of integral types or of references to Java objects

The following provides examples of different kinds of data:

```
int i=3453;  // integral type : integer
byte b=85;   // integral type : byte
bool o=true; // integral type : boolean

myColorClass Blue; // reference type
if (Blue.foo == true) // read a field using reference
   Blue.bar = 2;      // write a field using reference

int a[] = {1,2,3}; // array of integral type
myColorClass t[] = {Blue, Red, Green};
```

Assuming that the C component has a reference to the Java context ("`env`") and the object reference ("`obj`"), access to Java data is achieved through the following 3 steps :

- get a reference to the class of the object
- get a reference to the class to which the object belongs through a JNI service
- get a reference to the field of the object through a JNI service
- request the value of the object field through a JNI service.

In the following example, we want to access the field called "`foo`" of the following class from the native code

```
class myClass {
     int foo;
     native void myNativeMethod();
}
```

The corresponding C code looks as follows:

```
JNIEXPORT void JNICALL Java_myClass_myNativeMethod
     (JNIEnv *env,
      jobject obj) {

     // step 1 : get class
     jclass cls = (*env)-GetObjectClass(env, obj);
     jfieldID fid;
     jint value;

     // step 2 : get field id
     fid = (*env)-GetFieldID(env, cls, "foo", "I");
     if (fid == 0) { return; } // Check reference consistency

     // step 3 : get field value
     value = (*env)-GetObjectField(env, obj, fid);
}
```

4.6.1.5 JNI Access of C Data in Java components

Reciprocally, Java components may wish to access C structures.

The following types of data are defined in C :

- integral types (`char`, `int`, `long int`, ...). The possible operations are read and write.
- arrays of integral types or of C structures
- C structures

The following provides an example

```
// integral types
char c=32;
int i=12032;
long int l=543214;

// C structure
struct foo {
     int field1;
     char field2;
     long int field3;
} f;

// arrays
int a[3] = {1,2,3} // of integral types
struct foo s[2] = {{1234,4,453244}, {4321,5,-3556433}} // of structures
```

JNI does not support direct access to C data. Instead, it is expected that Functions (usually called accessors) are provided to read and write data.

4.6.1.6 JNI Handling Java Features in C components

C components can also handle some Java features.

Exception Throwing

This mechanism is interesting to propagate an error that occurred during the native code execution to the calling Java method.

In JNI, the C code calls a JNI function to create and throw a Java Exception object:

```
newExcCls = (*env)->FindClass(env,"myException");
if (newExcCls == 0) {
     /* Unable to find the new exception class, give up. */
     return;
}
(*env)->ThrowNew(env, newExcCls, "my Exception thrown from C code");
...
```

Exception Handling

This mechanism is interesting to treat an error that occurred during the execution of a Java method.

JNI gives some simple services to handle exceptions raises in a Java method call: the C code calls a JNI function to know if an exception has occurred:

```
...
exc = (*env)->ExceptionOccurred(env);
if (exc) {
...
```

Synchronisation.

The C code executes portions of code that need to be synchronised according to the Java synchronized semantics. In the usual JNI, this is done very simply with the `MonitorEnter`/`MonitorExit` services. For example, the following Java `synchronized` block:

```
synchronized (syncObject) {
// synchronised code
}
```

is synchronised with the following native block (supposing `obj` contains the reference to the same object as `syncObject`):

```
(*env)->MonitorEnter(env, obj)
// synchronised code
(*env)->MonitorExit(env, obj)
```

Object creation

The usual JNI is rich enough to allow you to create objects directly from the native code, through the `NewObject` service. Let's suppose you want to create an instance of the following Java class:

```
class MyClass {
     int foo;
     MyClass(int i) {
           foo = i;
     }
}
```

then, you will write a native code looking as the following:

```
jclass cls;
jmethodId mId;

cls = FindClass(env, "MyClass"); // obtain reference of the class
mId = GetMethodID(env, cls, "<init>", (V)I); // obtain constructor ID
obj = NewObject(env, cls, mId, 42); // allocate and construct object
```

which has exactly the same meaning as a Java code like:

```
MyClass obj = new MyClass(42);
```

4.6.2 Intended Use in Embedded Systems

4.6.2.1 Introduction

Typical Use

It is expected that future embedded systems applications will have well defined specific needs concerning native interface.

One case would be when the entire application is nearly entirely written in Java, with some parts remaining coded in C. Typically :

- All the application is written in Java, while run-time support (OS, drivers) is in C. Some part of the run-time could also be in Java, as this could be the case with drivers.
- Native interface is needed for specific purposes :
  - application call from C to Java in the following cases for instance :
    - the run-time calling application initialisation routines or starting Java threads,
    - interrupt routines calling some specific Java methods[8]
  - application call from Java to C for specific services such as IO services.

[8] Provided the Java code is compatible with timing constraints associated with the interrupt.

Therefore, involved C components usually need standard programming interfaces which are not specific to Java programming interfaces.

Typical Native Interface Profiles

It is consequently expected that embedded systems need a subset of the full set of JNI features. There could be different such subsets of profiles. For instance

- The full JNI profile. Native methods can mimic the entire semantics of Java. For instance a native method can raise exceptions, can create objects, can access objects fields and methods. In order to guarantee portability of native methods (for instance written in C), all information about interface parameters are obtained and described dynamically in the form of strings.
- Full Java support with no dynamic discovery of interfaces. The full semantics of the Java language is supported. On the other hand dynamic discovery of interface parameters is not supported. Instead, some other binding techniques such as configuration directive at the development kit level. This approach is obviously dependent on the underlying development platform. It can only be portable if a consensus for a standard is reached.
- C functions support with no dynamic discovery of interfaces. Native methods only support parameters types that are defined in the C programming language. Supported parameters could be integral types, or possibly reference to objects, in the case the use of references is limited (e.g. to storage)

Typical Constraints

As, embedded systems have specific requirements in terms of initialisation and generation, real-time constraint, memory needs which may influence native interface mechanisms :

- It is often preferred or even required in the case of static systems that no dynamic initialisation takes place
- The native interface should have real-time behaviour if used in portion of codes that have real-time constraints
- Memory constraints could also have an impact (for instance it would be too costly to declare strings in the code).

4.6.2.2 Mechanism to Call C

The selection of a suitable mechanism to perform the calling of a Java native method depends on two aspects, the binding mechanism and the parameter types

Binding Mechanism

A mechanism is needed to associate a Java native method to a C function. Many approaches are available. They depend on decisions taken at the level the development tool and specifically the compiler, and on criteria such as portability or performance . Let us take two examples :

The compiler could generate a call to one single C function (let us call it `NativeMethod`) with a parameter which identifies the native method. This function will then retrieve the function

addresses and the right parameters in order to call the final C function (let us call it `Foo`). A service is available to the `NativeMethod` C code to retrieve the C function address and related parameters (let us call if `Retrieve`). Here is the resulting code

```
Void nativemethod(tNativeMethodId NativeMethodId) {
      Retrieve(NativeMethodIf, &FunctionAd, &ParameterAd);

      /* call function */
      *FunctionAddress(ParameterAd)
}
```

The performance of this mechanisms is influenced by the following :

- Overhead to update the table. The table could be updated at startup time or just before invoking `NativeMethod`.
- Overhead to search the table when `Retrieve` is called. If the key is the method name that some hash table search could be used. While a hash table would yield a very efficient O(1) type of performance, it can lead to a O(N) worst case performance, something not acceptable for embedded systems with real-time constraint. On the other hand, if the key is an index, then a direct indexation with an O(1) performance is guaranteed.

This first approach is suited to real-time systems which do not require static initialisation, if O(1) performance is guaranteed in the call mechanism

A second approach could be the following : the compiler generates a call to a specific C function with the right parameters. The association between the Java native method and the C function is done at generation time by the compiler or/and associated tools. Again there are several possibilities :

- The association could be explicitly specified by the programmer, e.g. through a configuration file given as an entry for the code generator (for each C function, the name of a Java method is given).
- The association could be implicit as it relies on common conventions between the programmer and the code generator provider. Example: use of naming conventions (a static method is declared that can easily be deducted of the name in C language).

This second approach is suitable to embedded static systems.

Parameter Types

As stated in the introduction, it is believed that only standard parameters are needed :

- integral types are straightforward to support because of the direct correspondence with C types.

- Reference to arrays could be useful so that a C function can read and modify their content[9].
- Storing and object reference and returning it could also be useful. Accessing fields of an object is fairly complex, but as we already said, probably not necessary.

4.6.2.3 Mechanisms to Call Java

The selection of a suitable mechanism to perform the calling of a Java method from a C function depends on two aspects, the binding mechanism and the handling of virtual methods.

Binding Mechanism

Similarly to the section describing the call from Java to C, approaches to associate a C function to a Java method will depend on the level of handling performed at generation time versus at run-time.

We saw that an approach such as JNI (see 4.6.1.3) involves 3 service calls to get a reference to the class of an object, then to get the method of an object and then to invoke the method. This approach is suited to real-time systems provided internal search algorithms are achieved in O(1) performance. This is generally not the case (we rather have O(N) worst case).

Alternatively an approach involving configuration data could eliminate search algorithm. This is the recommended approach for embedded static systems. Note that a Java compiler would have to know that a given Java method can be called from C in order to avoid some compiler optimisation such as inlining some method: a sufficient property would be to declare such methods as `public`.

Virtual Method

A mechanism for virtual method call, i.e. invoking the right method in the inheritance scheme is involved. In JNI, a service to obtain a method identification is available once the class of the object is known (see 4.6.1.3, service `GetStaticMethodId`). Other implementations would have to provide a similar service or possibly access to the virtual method table.

In real-time systems, it would be desirable that the overhead of such search be bounded. In embedded static systems, it would be expected that no such search is performed, and that rather a static association be calculated. Such static association could be obtained as follows

- Possible ambiguities on which method implementation is to be called in an inheritance tree is removed, for example: only `final static native` methods can be called and they shall have a body. The programmer shall call them directly using the most precise class which implements the method. This can lead to very efficient implementation (no search for the method).
- Some programming practices are enforced in the use of inheritance so that the right method

[9] This functionality, if allowed, may have an impact on the compilers optimisation capabilities: the compiler should be aware of the possible modification of data from C so that he will not make wrong assumptions on data and keep out-of-date values.

can always be determined statically, such as only allowing static methods. Note that enforcing static call with no inheritance is not a real restriction as the Java programmer could introduce some code to allow for such operation. For example, assume that he wishes to call the `bar` method of the following class:

```
class Foo {
     int bar(byte a);
}
```

In order to achieve this, the programmer can introduce the following class:

```
class MyStubs {
     static int stub_bar(Foo f, byte a)
         {
            return f.bar(a);
         }
}
```

Then he can access the `bar()` method of a `Foo` object `myfoo` via the following call[10] :

```
result = MyStubs.stub_bar(myfoo,a);
instead of:
result = myfoo.bar(a);
```

4.6.2.4 Accessing Java Data in C Components

The selection of a suitable mechanism to perform access to Java Data in C components depends on the following aspects, matching data format, managing object referencing, and managing access to shared data.

Matching Data Format

Matching of Data Format is not straightforward because (1) Java does not specify the physical representation of its objects and (2) C does not specify it either (even though the underlying physical representation is generally fairly straightforward).

The following issues have to be taken into account :

- Java integral types data representations are well defined, i.e. Java basic types have a fixed length whereas all the C types can differ from one platform to another.
- Mapping of structures is not straightforward. Padding issues will typically take place as it will probably the case in the following example.

[10] Note that even if though this technique does not look convenient, JNI programmers are used to it.

```
struct Date {
     byte    day;
     int     year;
}
```

This is the reason why JNI proposes a procedure interface where each access necessitates 3 calls (see 4.6.1.4). While this is guarantying portability of C components, this approach does not allow for efficient access to data. In embedded systems, it would be more appropriate to have a Java compiler which is well integrated with a C compiler so that the matching of data formats is straightforward.

Managing Object Reference

The fact that a C component may use an object reference may have an impact on the way such references are handled in the Java run-time in terms of object reference lifetime. The fact that a C component is using or event storing an object reference must be known to the Java run-time so that the object reference does not disappear (e.g. because the reference is no longer used in the Java program, and consequently the object has been reclaimed).

In JNI, the fact that object references are transmitted as parameters guarantee that during the call, the object will not be garbage collected. On the other hand, if the C component stores the reference for future use and returns, then there is an issue of marking the reference. This would be possible if some native interface primitive is available for this.

Access to Shared Data

The storage of object reference may jeopardize data integrity is concurrent access may occur, from a Java thread or interrupt routine on one side and from a C thread or interrupt routine on theother side. To handle that, C components should have access to the mutual exclusion services used internally by Java threads.

4.6.2.5 Accessing C Data in Java Components

The selection of a suitable mechanism to perform access to C Data in C Java depends on the mapping approach, the matching of data format, and on other implementation considerations.

Overall, allowing access to C data in Java component could have an effect on portability of C code.

Mapping

Several approaches are possible to make C data accessible to the Java world:

- Access the variable via a native function. This solution has a performance cost that could be incompatible with embedded systems requirements, unless C macros are used instead of C procedures. Such access could be achieved as follows :

- Access to the C variable through a C function written by hand by the developer: this is the simplest solution. This is the approach used by JNI.
- Automatic generation of the desired access functions at compile time. This necessitate support from the development tool.

▪ Make the C variable appear like a Java "variable" or object. Different solutions can be also be considered such as :
- One object created per C variable. This approach could be an overkill in many embedded systems. It is not even obvious that this kind of solution would be wise in terms of software engineering, except in specific cases such as accessing a C array of bytes as a Java array of bytes.

```
byte tab[12];
tab = arrayAccess(C_tab_address);
```

- One particular Java class is created which has one field per C variable. This approach look reasonable

Matching Data Format

See analysis in 4.6.2.4.

Implementation Considerations

Other implementation aspects would have to be taken into account

▪ Type of C data shared. It is likely that only static C data is made available (local variables for example are not meant to be accessible)
▪ A straightforward manner to match simple data types. For instance a convention could consist in declaring all the accessible native data with special types such as “jint”, “jbyte”…
▪ Access mechanims. C data could be accessed in the Java world through some kind of “proxy” (i.e. a Java variable which represent the native one).

4.6.2.6 Handling Java Features in C Components

Exception Throwing

Allowing C components to throw exceptions implies the availability of some run-time primitives. It could be assumed that such exception are intended to be caught by Java code. Some programming guidelines would have to be provided. JNI is providing such feature. In embedded systems, the feature would depends on the run-time model for exception handling. It could thus prevent C component to be portable (unless some standard for this emerge). Further, the verification that the C component conforms to guidelines for exception throwing is not obvious (maybe it could require a preprocessor at the C level).

Exception Handling

Concerning exception catching, the need for this functionality has to be well evaluated. It could be assumed that because of the limited call from C to Java, support will be limited to the following :

- when the underlying OS has started a thread of called a method, then it will catch, all implicit exceptions
- when an interrupt calls a method which generate an exception, then it will catch all implicit exceptions.

In other words, the C code does not propagate exceptions. If it was not the case then further issues would have to be solved such as :

- using a mechanism to search for the right `try` block. We saw in the section on exception that this mechanism is compiler dependent.
- distinction possible between "Unchecked" (throws clause is not mandatory) / "Checked" exceptions

Supporting Synchronisation

Concerning synchronization, it is expected that the underlying OS offer services to handle synchronized methods. This would allow C threads and Java threads to concurrently access synchronised methods. JNI is providing such features. In embedded systems, the feature would depends on the run-time API. For instance in OSEK one could use the GetResource and LeaveResource primitives. Unless some agreement on the API, it would prevent C components to be portable.

4.6.3 Recommendations

4.6.3.1 Mechanism to Call C

Concerning binding, the approach where the called C function is automatically deducted and statically bound by the Java compiler is preferred.

Concerning parameter type, it is recommended to support

- integral types
- object references for storage and restitution
- arrays of the above types:

4.6.3.2 Mechanism to Call Java

It is recommended to use a binding mechanism based on configuration data.

It is also recommended that only call to static method are allowed in order to avoid virtual method search

4.6.3.3 Accessing Java data in C Components

It is recommended not to use it, because of the high cost involved in C components which are typically intended to be constrained (in performance or in memory).

4.6.3.4 Accessing C Data in Java components

If the application makes extensive use of system variables, I/O ports, or any memory area with a fixed address for example, the Java to C compiler should support an efficient access to data, at least for integral types and arrays of integral types. A possible solution would be the use of C macros.

4.6.3.5 Handling Java Features in C Components

Throwing Exception

Exceptions throwing in C code should be supported. Programming guidelines should be provided. APIs should be provided by the underlying run-time. As this API depends on the compiler, it is recommended that some standardisation takes place.

Handling Exception

It is recommended to support exception catching but not exception propagation.

Synchronisation

An API should be provided to support synchronisation between C and Java threads. This API should be independent from the underlying OS.

5. Lessons Learned, Recommendations

The AJACS project allowed an in-depth analysis of the feasibility to use Java for embedded systems. We now have a number of recommendations to provide to the Java and embedded systems community in order to be able to use Java for embedded systems in the future. Some of the recommendations were not followed by AJACS, for a number of reasons :

- Recommendations concern topics that were not covered by AJACS such as debugging
- Recommendations concern issues that were discovered within the project such as the necessity to have a preprocessor or similar mechanism for system programming
- Recommendations concern return on experience from the project which made the consortium change its opinion

The below tables summarize those recommendations.

Development Aspects		
Type of Issue	**Recommendation**	
Development Environment	Change the semantics of root classes Do not use standard libraries Have a preprocessor or suitable compiler features Have a debugging interface	
System Programming	Extension/Support of constants, controlled inlining, unsigned types, bit management Extension to support interrupt management	

Real-Time Support		
Type of Issue	**Recommendation**	
Predictability	No garbage collection Programming practice to limit inheritance	
Synchronization	Change the Java Memory Model Use synchronized for locks Support locks in interrupt routines	

Support of Exception		
Type of Issue	**Recommendation**	
Software Engineering	Structured programming validation Default handler validation	
Real-Time Behavior	Exception handling overhead validation Exception handling response time validation	
Memory Needs	No exception instantiation in the throw statement of applications Likewise for run-time	

Initialization		
Type of Issue	**Recommendation**	
Class Initialization	Download all classes at compile-time	

	Execute class initialization during the startup phase / Handle class dependencids	
Object Initialization	Predefined object approach	
Constants in ROM	Further investigation Neede	

Memory Management		
Type of Issue	**Recommendation**	
Limited Amount of Memory	In case downloading is needed use JEFF	
Static Systems	Use Native Code Programming practices for object creation JVM traceability Objects in stacks See initialization	
Hard Real-Time Systems	See garbage collection Use Memory Banks	

Native Interface		
Type of Issue	**Recommendation**	
Mechanism to all C	Automatic binding Parameters limited to integral types, object references and arrays of integral types and object references	
Mechanism to call Java	Binding mechanisms based on configuration data Call to static methods only	
Accessing Java Data	Not recommended	
Accessing C Data	Some efficient access should be made available possible through C macros	
Handling Java features in C	Exception throwing in C should be supported Exception catching in C should be supported Synchronisation between C and Java should be supported	

6. References

[D6] AJACS Architecture definition version 1.0. D6 deliverable. AJACS Consortium, June 14th 2001.

[Gupta02] Rajesh Gupta. U.C.Irvine. Winter 2002. "http://www1.ics.uci.edu/~rgupta/ics212/w2002/intro.pdf"

[Hoare81] The Emperor's Old Cloths. The 1980 Turing Award Lecture. C.A.R Hoare. CACM. February 1981

[Java Language Specification]

The Java Language Specification. Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. Addison-Wesley, 2000.

[JEFF] JEFF File Format Specification v1.0. J Consortium. March 2002. ISO/IEC 20970.

[JNI] Essential JNI: Java Native Interface. Rob Gordon. Prentice-Hall, 1998.

[JVM Specification] The Java Virtual Machine Specification. Tim Lindholm, Frank Yellin. 1997. Addison-Wesley.

[Lea 96-99] Concurrent Programming in Java: Design Principles and Patterns. Doug Lea. 1996-99. Addison-Wesley.

[OSEK] OSEK-VDX real-time kernel standard. "http://www.osek-vdx.org"

[Pugh] Fixing the Java Memory Model. Bill Pugh.1999. "http://www.cs.umd.edu/~pugh/jmm.pdf"

[Real-Time Core Specification] "Real-Time Core Extensions for the Java Platform - Draft 1.0.14" .J Consortium. May 30, 2000. This document is available on the website of the J consortium : "http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf"

[RTSJ] JSR-000001 The Real-Time Specification for Java. The Real-Time for Java Expert Group. 2001. Addison Wesley. This document is available on the website of the Real-Time for Java Expert Group: "http://www.rtj.org/rtsj-V1.0.pdf"

[Sun exception] What's an Exception and Why Do I Care? "http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html"

[Venners 98] Object Initialisation in Java. Bill Venners. JavaWorld, March 1998.