View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Lund University Publications - Student Papers

ISSN 0280-5316
ISRN LUTFD2/TFRT--5801--SE

Safe Programming Languages for ABB Automation System 800xA

Markus Borg
Lukasz Serafin

Department of Automatic Control
Lund University

August 2007

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
August 2007

Document Number
ISRNLUTFD2/TFRT--5801--SE

Author(s)
Markus Borg and Lukasz Serafin

Supervisor
Ola Angelsmark at ABB in Malmö and Anders Nilsson at Computer Science in Lund.
Karl-Erik Årzén at Automatic Control in Lund (Examiner)

Sponsoring organization

Title and subtitle
Safe Programming Languages for ABB Automation System 800xA (Säkra programspråk för automationssystemet ABB 800xA)

Abstract
More than 90 % of all computers are embedded in different types of systems, for example mobile phones and industrial robots. Some of these systems are real-time systems; they have to produce their output within certain time constraints. They can also be safety critical; if something goes wrong, there is a risk that a great deal of damage is caused. Industrial Extended Automation System 800xA, developed by ABB, is a realtime control system intended for industrial use within a wide variety of applications where a certain focus on safety is required, for example power plants and oil platforms. The software is currently written in C and C++, languages that are not optimal from a safety point of view. In this master's thesis, it is investigated whether there are any plausible alternatives to using C/C++ for safety critical real-time systems. A number of requirements that programming languages used in this area have to fulfill are stated and it is evaluated if some candidate languages fulfill these requirements. The candidate languages, Java and Ada, are compared to C and C++. It is determined that the Java-to-C compiler LJRT (Lund Java-based Real Time) is a suitable alternative. The practical part of this thesis is concerned with the introduction of Java in 800xA. A module of the system is ported to Java and executed together with the original C/C++ solution. The functionality of the system is tested using a formal test suite and the performance and memory footprint of our solution is measured. The results show that it is possible to gradually introduce Java in 800xA using LJRT, which is the main contribution of this thesis.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
56

Recipient's notes

Security classification

http://www.control.lth.se/publications/

Acknowledgments

We have had a great time working with this thesis and would first of all like to thank our supervisor at ABB in Malmö, Ola Angelsmark, for always having time to answer our questions right away, no matter how much more important things he had to do. We also would like to thank our two supervisors in Lund, Anders Nilsson at the Department of Computer Science and Karl-Erik Årzén at the Department of Automatic Control for answering our questions and providing us with valuable feedback.

Finally, thanks to all people working at ABB for making us enjoy our time as thesis students. We are grateful to Adam, Anders and Markus for helping us with practical details and a special thank you to Jan and Robin for everything you did for us regarding the weekly floorball!

Lund, August 2007

Markus & Lukasz

Contents

Abstract 1
Acknowledgments 2
List of Acronyms 7
1. Introduction 8
1.1 Problem description 8
1.2 Method 9
1.3 Previous work 9
1.4 Limitations 10
1.5 Thesis outline 10
2. Requirements on programming languages 11
2.1 Productivity 11
2.2 Safety 11
2.3 Concurrency 13
2.4 Determinism 13
2.5 Memory footprint 14
2.6 Performance 15
2.7 Portability 15
2.8 Other factors 15
2.9 Summary 17
3. Languages for safety critical real-time systems 18
3.1 C/C++ 18
3.2 Java 21
3.3 Ada 26
3.4 Summary and discussion 28
4. Description of 800xA 31
4.1 Overview 31
4.2 Redundancy 32
4.3 Role Selection 33
5. Introducing Java in 800xA 35
5.1 Porting the module Role Selection 35
5.2 Including Java threads 40
5.3 Difficulties encountered and lessons learnt 41
6. Results 43
7. Conclusions 46
7.1 Future work 47
8. Bibliography 48
A. Specific modifications to integrate Java 51
A.1 Details for Role Selection 51
A.2 Details for Java threads 53
B. LJRT configuration file 54

List of Figures

2.1 Example of ambiguity in C++. 12
2.2 Relative and absolute time delays 14
4.1 Schematic figure of a single PM. 31
4.2 Schematic figure of a two connected PMs. 32
4.3 Example of takeover following a failure in the forth execution unit. 33
5.1 Our solution using a bridge for calls to and from the 800xA source code. 39
5.2 Calls from the 800xA system to our generated C code. The dashed lines represent returns from calls. 40
5.3 The two Java threads we run in the system. 40
6.1 The memory situation after the startup sequence. 45
7.1 The development cycle. 47

List of Acronyms

API Application Programming Interface
An interface that a system provides to support requests for services

CEM Communication Expansion Module
Communication module for 800xA

GC Garbage Collector
A collector that attempts to reclaim memory that will never be accessed

GNU Real-Time Operating System
An operating system intended for real-time applications

ITC Initialization-Time-Compilation
Compiling code at program start

JIT Just-in-Time compilation
Converting program code from one format into another during runtime

JNI Java Native Interface
A framework that allows Java code to call and be called by native applications

LJRT Lund Java-based Real-Time
A solution to run Java on limited platforms

PM Processing Module
The actual 800xA hardware unit

POSIX Portable Operating System Interface
Standards to define the API for software compatible with variants of Unix

RTOS Real-Time Operating System
An operating system intended for real-time applications

RTSJ Real-Time Specification for Java
A solution to use Java in real-time systems

VOS Virtual Operating System
ABB internal interface between VxWorks and 800xA

WCET Worst Case Execution Time
The worst case execution time of a task

1. Introduction

The computer industry changes rapidly, as does the real-time computing subset. The introduction of embedded systems in everyday applications such as TV sets, mobile phones, and microwave ovens makes us continually more dependent on real-time software. Numerous embedded systems are also safety critical, such as the ones used for aircraft, satellites, automation industry and medical applications.

The influence of those systems with safety related applications is steadily increasing and being able to master them safely with a sufficient degree of confidence is a most important challenge. A common situation in the industry today is that the verification process is the major developing cost of critical systems. Enormous effort is needed to, through reviews and testing, verify the behaviour of the developed software to ensure that it meets the industrial demands.

Another aspect is that the complexity of embedded applications rises as well. More and more functionality is required by the software, instead of as traditionally implemented in hardware. It is easier, faster, and more flexible which is beneficial to all market actors. As complexity grows, low-level programming is no longer productive enough, thus real-time systems need to be developed using higher-level languages in order to be cost efficient.

Industrial^{IT} Extended Automation System 800xA is a product developed by ABB, whose purpose is to act as an advanced process control system. It is operating in various areas in the industry, for instance in the chemical industry, pulp and paper industry, in power plants, and oil platforms. 800xA was especially developed for use in such critical domains and common to those environments is that there are high demands on safety and availability. In those situations the impact of a system failure can cause major monetary losses and also endanger human lives.

1.1 Problem description

The 800xA software is currently consistently written in C/C++, languages which give the developers extensive freedom. This freedom is not optimal from a safety viewpoint and consequently much effort and money must be spent on verifying the behaviour of the code. The cost of the testing and the many reviews could be reduced by introducing a language which prevents many of the common programming errors from being made.

The objective of this work is to determine whether there are any plausible alternatives to the current implementation languages. Our first step will be to conclude what is needed from a language used for developing safety critical real-time software. This will be followed by an evaluation of the candidate languages we chose and a comparison with the C/C++ solution of today. Then we intend to more practically test an alternative implementation language. This will be done by porting a part of the 800xA system to Java and observing if any performance differences can be noticed.

The topic of this thesis can be stated by posing the following questions:

1. What is required from a real-time language suitable for developing 800xA?
2. Which current programming languages offer those features?
3. Is it possible to use Java to implement a part of 800xA?
4. Will any significant performance differences be encountered?

1.2 Method

During the first phase of this project, we will examine relevant literature in order to identify characteristics needed from real-time programming languages. We will look into sources of various origins, both from the academical and industrial sphere, and our aim is to provide an as unbiased summary as possible.

Our next step will be to assess our candidate languages on the basis of the found factors. The findings we consider being of most importance will be summarized and presented in a separate chapter.

For the practical part of our work, we will use a solution called Lund Java based Real-Time (LJRT) [Pro07]. It has been developed at the Department of Computer Science at Lund Institute of Technology and it is presented further in Section 1.3 and 3.2, and more extensively in [Nil06].

1.3 Previous work

Language design has been a very active research area and the choices of languages to use for implementing real-time software have changed accordingly. The demands on the languages have also changed significantly and in the second chapter of *Real-Time Systems and Programming Languages*, Alan Burns and Andy Wellings identify three classes of languages historically being used [BW01].

- **Assembly languages**. In the beginning of this engineering discipline, almost all real-time systems were implemented in the assembly language of the embedded computer used. This machine-oriented approach does not provide any direct benefits, but during the first years this was the only way to achieve efficient implementations that also could access the needed hardware.
- **Sequential languages**. In the 1960s, the programming languages had become much more mature at the same time as computers were powerful enough to really take advantage of higher level languages. The languages then used were all sequential and did not support any real-time constructs and instead had to rely on the underlying operating system. Examples of the languages used were Jovial developed in 1959 and Coral 66 developed in 1966, both strongly connected with military applications. In recent times, C has been the most popular and successful, developed by Dennis Ritchie in 1972.
- **High-level concurrent languages**. In the 1970s, computer-based systems began to get bigger and more complex. Developing using the existing languages was no longer suitable, since it was expensive and inefficient to develop and maintain sophisticated software. Many languages emerged, aimed at aiding development of embedded real-time systems. Examples include Modula-2, invented by Niklaus Wirth around 1978 which contained a useful module system. Ada was released in 1983 and a version with support for object-oriented programming, Ada95, was released in 1995. C spawned many new variants and the most important, C++, has support for object-oriented programming but is without language constructs for concurrency. Java has those features and quickly became popular, much because of the Internet.

There has been much research conducted on the possibility to use Java in the real-time domain. Relevant recent work include Nilsson, Ekman and Nilsson [NEN02], which presents the advantages of using Java and some of the pitfalls involved. The

paper offers a good overview of the topic of this thesis. Lund Java Based Real-time (LJRT) is presented in Nilsson [Nil06] and describes a complete solution for executing Java on very limited platforms, using Java-to-C compilation. Robertz [Rob06] discusses automatic memory management in the domain of real-time computing and complements the work of Nilsson. Another approach to run Java on restricted platforms is to use tailor-made virtual machines. Ive [Ive03] presents such a solution. Some relevant master's theses using virtual machines have been finished at the Department of Computer Science in Lund lately. One of those is Andræ and Gustavsson [AG02], where the authors used Ives solution together with the 800xA system. Our work, however, will focus on lower level parts of the same system. Arvehammar [Arv07] used the same approach, but another virtual machine, the *JamaicaVM*, to be able to run on an embedded system. JamaicaVM is an implementation of the *Real-Time specification for Java* (RTSJ) [BBD+00] produced by the Real-Time for Java Expert Group. RTSJ identifies a number of areas, where Java must be enhanced in order to make Java useful in real-time systems. There are also other implementations of the specification on the market, for instance *Mackinac* [Mic04] by Sun Microsystems.

The main novelty of this thesis is the application domain, introducing another language in the lower level parts of the control system 800xA.

1.4 Limitations

Many programming languages are used within the domain of real-time computing and it would not be possible to include all of them within the scope of this thesis. We have decided to limit the examination of languages to the three candidates we consider most relevant.

C/C++ is the current language solution and is consequently needed for the comparison. *Ada* is part of the study since it was designed for safety critical systems and has been heavily used within certain areas. Due to the limited time frame of this thesis work, we have decided to assign a lower priority to this language and its part in this study is as a result less extensive. *Java* is the third language, a language that is a strong trend in the industry and on which much research is currently conducted. This limitation enables us to make an as complete assessment as possible.

1.5 Thesis outline

The rest of this report has the following outline:

Chapter 2 presents what is needed from a programming language.
Chapter 3 describes the currently used languages and makes a comparison with Java and Ada.
Chapter 4 gives a brief overview of the control system 800xA in general and the module we are working with in particular.
Chapter 5 describes our work with introducing Java in 800xA.
Chapter 6 presents the results we obtained by testing our solution.
Chapter 7 finally concludes our findings and discusses the contributions of this thesis.

2. Requirements on programming languages

To constitute a usable real-time alternative, a programming language must have certain characteristics and provide support for some necessary functionality. This is especially true in the domain of safety critical systems. We have identified the following main factors: productivity, safety, concurrency, determinism, memory footprint, performance and portability. In Chapter 3 we evaluate how well they are supported by the languages in our study.

2.1 Productivity

The strong trend in the embedded system industry is that more of the functionality is implemented in software instead of hardware. As a result of that, the functionality gets more and more complex [Mic04]. This causes a loss of programmer productivity if the language does not provide enough support for *abstraction*[1]. A result of this is that object-oriented languages with built-in mechanisms such as classes, inheritance and virtual methods get more interesting for the development of real-time systems [Nil06]. The fact that the design mirrors the problem domain simplifies the design cycle and the system gets easier to understand and maintain for new developers [Eve07b]. Language features that support decomposition and abstraction are said to aid *programming-in-the-large*. If the language offers good support for modularization, the possibility to reuse parts of the software increases. Reusing software greatly increases productivity, assuming it is managed in a good way.

The possibility to find errors at an early stage also serves the interests of productivity, a chance that is increased by using a high-level language. Working with languages that offers a high level of abstraction even precludes some errors from being introduced at all [Rob06]. For example, problems with manual memory management are not an issue in languages with built-in *garbage collection*. The errors that despite everything occur should be presented to the developer with as descriptive error messages as possible, to enable quick and efficient correction.

Another factor that strongly influences productivity is *simplicity*, a good characteristic of any design. Avoiding a too complex language solution reduces the cost of programmer training and limits the risks of making errors because of misunderstandings of the language constructs [BW01]. This is closely related to safety and it is covered further in Section 2.2.

Last but not least, programmers' productivity is increased if they have experience of the language. Therefore, well-known languages are preferred to languages that few programmers have experience of.

2.2 Safety

If something goes wrong in a safety critical system, there is obviously a risk that a great deal of damage is caused. This risk means that the number of software errors in the system should be kept at an absolute minimum, which in turn makes demands

[1] Reducing details so that the developer can focus on concepts.

on the implementation language. The language should first and foremost *preclude errors*. If this is not the case, errors should be detected as early as possible, preferably during compilation (an example of this is when the compiler tells the programmer that a variable has not been initialized) or at last by the language run-time system. Detecting errors at compile time is safer since they have to be removed before there can exist an executing version of the system. All possible execution results should be clear from reading the source code of the program [Nil06].

A language obviously cannot find logical mistakes in the program. Instead, for a language to be considered safe, it should have a *consistent design* and be *well structured* [BW01]. Another safety related demand you can put on a language is that it should not include any ambiguities [BW06]. Ambiguities exist for instance in C++. Consider the code given in Figure 2.1. The meaning of this code is not unique, given only this fragment. If `b` is a value or variable, the integer `a` should be initialized to the value `b`. If `b` is a type, the statement is a function declaration and `a` is a function with an argument of type `b` returning an int. The absence of ambiguities combined with a *high readability* should make errors easy to notice and to identify in already existing code.

```
void f() {
  int a(b);
}
```

Figure 2.1 Example of ambiguity in C++.

High readability requires among other things reasonable keywords and the possibility to modularize the program. A good language should enable the reader to understand the operation of a program without needing to study flowcharts or additional documents [You82]. Languages should for example not restrict the length of identifiers. The use of a separator should be allowed to enable complex names, for example `temp_max`. The space character is however not an acceptable separator, which was a poor lexical convention of the language FORTRAN, causing many unnecessary errors [BW01].

A program is probably read many more times than it is written, which means the implementation language should to a reasonable extent favour the reader rather than the writer. Again meaningful names of keywords can be used as an example.

There might be a trade-off between focus on safety and simplicity. A safe, easily readable language can reduce documentation costs and improve the error detection rate but may result in a too complicated language. This can for instance increase the length of the source code and increase compilation time and threaten productivity, making it necessary to find a suitable balance between the possibly contradicting aspects [BW01].

A safe programming language should be well-defined. The standard of the language should for example tell what will happen if division by zero occurs. If this is not the case, the behaviour of programs may vary across platforms, and errors may be harder to detect.

2.3 Concurrency

The real world is parallel. Therefore, virtually all real-time systems are concurrent. A lot of events can occur at the same time, and the work that has to be done to handle each event constitutes a task. When we have a limited number of CPUs, we have to map the tasks into concurrent processes that share the available CPUs. This programming paradigm is called *concurrent programming* [Årz03]. Without support for concurrency, the software must be constructed as a single loop. This makes it extremely difficult to reliably write code satisfying timing requirements.

Concurrent processes are rarely totally independent of each other. They need to communicate with each other. When communicating, they need to coordinate — this is called *synchronization* [Årz03].

Threads are processes that run within the same ordinary address space. A programming language used for real-time systems should have a built-in threading model and support thread synchronization. If the language does not support concurrency, the programmer needs to use an external application programming interface or a real-time operating system. This can compromise portability [Bro07], described in Section 2.7.

2.4 Determinism

For a real-time system, it is not sufficient only to produce correct output; it also has to be within certain time constraints. This is especially vital to the hard real-time systems this thesis is aimed at. For any safety critical system of this type, the worst case response times of the involved tasks must consistently be shorter than their corresponding deadlines [Årz03].

The worst case execution time (WCET) of a task is the maximum length of time the task could take to execute. There are two approaches to obtain an estimate of the WCET, measurements or analysis. The obvious drawback of measuring execution time is that it is very hard to be certain that the worst case has been observed. Analysis on the other hand, suffers from the fact that modern processor architectures are complex, using caches, pipelining, branch predictors etc. To predict the effect of all features is hard and to ignore them results in too pessimistic estimates. Finally, all loops and recursions must be bounded to make static analysis possible. Given the considerable amount of testing that real-time systems are subject to, measurements is the common industrial practice [BW01].

In order to have the maximum level of determinism, it has to be possible to compute the WCETs of all the various tasks. Most programming languages provide an amount of memory, the heap, for programmers to use during run-time. If the allocated space is released automatically using automatic memory management (garbage collection), this can have a serious impact on the ability to analyse the WCET. Most implementations of garbage collectors are not at all deterministic and significantly influence the execution times of time-critical tasks in ways hard to predict. Research has however shown that *deterministic automatic memory management* indeed is possible, but there is still a reluctance to rely on this in the industry of time-critical systems [BW01].

Scheduling is the ordering of tasks performed by the real-time system. The tasks must be ordered in such a way that they meet their deadlines. If there is no ordering that enables this, the system is said to be unschedulable. There are many different scheduling approaches. The most commonly used is preemptive fixed priority

scheduling [Årz03]. With this approach, the tasks have fixed static priorities which determine the order the tasks are executed in. If a low-priority process is running and a task of higher priority gets ready to run, there should be an immediate switch to the task with higher priority [BW01].

A programming language for real-time systems should have facilities for *assigning priorities* to tasks and for establishing appropriate *scheduling behaviour* [Bro07]. It should also be possible to access a real time clock, and it should be possible to delay until a specified time before continuing execution [Whe07]. Why this is important is explained in Figure 2.2. Assume that PerformAction2 should occur 5 seconds after PerformAction1. The problem with the the left code example is that it might be preempted after the value `5.0 - (getTime() - startTime)` has been calculated, resulting in a longer delay than wanted. This is not the case in the right example, where waitUntil is accurate at least in its lower bound.

Relative time	*Absolute time*
startTime = getTime()	startTime = getTime()
PerformAction1	PerformAction1
wait(5.0 - (getTime() - startTime))	waitUntil(startTime + 5.0)
PerformAction2	PerformAction2

Figure 2.2 Relative and absolute time delays

2.5 Memory footprint

Embedded systems usually run on very constrained platforms, for this reason it is necessary to minimize the memory usage. This essential requirement means that the solution to use has to have a *strictly limited memory overhead* and the developers has to manage with a small heap [Ive03].

The absolute majority of the computer systems world wide are embedded and they are typically mass produced for a specific purpose. The amount of memory available can vary, but the struggle for cost efficient products often results in evident limitations. On the other hand, the development costs tend to decrease when more memory gets available, since high-level languages can be introduced, which increases productivity (see Section 2.1). In the end, one has to compromise and weigh the arguments for and against more or less memory [NEN02].

A solution that is used in some cases is to utilize a virtual machine that creates an environment with certain properties between the computer platform and its operating system. In this case it isolates the application from the computer and lets it run using an interpreter or so called “Just-in-Time compilation” (further described in Section 3.2 under Performance). A typically known application is the Java Virtual Machine [LY99] developed by Sun Microsystems. Virtual machines require various amounts of memory and efforts have been made to tailor versions for embedded real-time systems.

2.6 Performance

The constraints on performance of embedded systems are similar to the constraints of the memory footprint. In order to produce cost efficient systems, normally only *very limited processors* are available. Another reason for the limited performance is that the systems frequently operate in environments where the power consumption is limited and not allowed to pass a certain threshold. This makes several processors inapplicable [Nil06]. Ultimately, the cheapest component able to do the job is often chosen.

2.7 Portability

If a program can be taken from one machine, compiled without change on another machine and run without any change in the output, it is considered fully portable [Hat95]. Of course, the situation is even better if the program does not even need to be recompiled.

A high level of portability is always a desirable characteristic of software developed for embedded systems. The main reason is that the same solution can be used on new hardware when it is needed and also that the real-time operating system used in the end product later might not be known during the development stage. Another advantage of good portability is to be able to develop the system in a suitable development environment and then run it on the required platform without having to change the source code [Nil06].

For the language to be considered portable, it should be flexible enough to allow the developer to express the required operations without having to resort to commands of the operating system or machine specific code to achieve the results [BW01].

Portability is closely related to some other requirements. If the memory footprint of a program is large, it may not work correctly on systems with little memory. If portability is bad, and the program is intended for many different systems, safety is affected. Therefore, a safety critical system which must be portable should not depend on implementation-defined or other non-portable features [Hat95].

2.8 Other factors

When studying high-level languages, it is useful to distinguish between the characteristics that assist the high-level design of complex systems and the actual implementation of functionality on lower level. This later set of features is called support for *programming-in-the-small* [BW01]. This area has been studied for decades and the languages we study have many similarities, the languages are for instance all block structured, they have similiar dicrete types (see Table 2.1) and both Java and Ada are strongly typed and assignments and expressions must involve the same type [BW01].

Furthermore, all the languages require sequential execution, allow recursive procedure calls and have very similar control structures for decisions and loops. One difference is that Ada and C have the unfavourable `goto`[2] statement, thoroughly discussed by Dijkstra in [Dij68]; another is that Ada has restricted the use of the loop variable, which can be modified in Java and C.

[2] `goto` is a reserved word in Java, but it does not serve any function.

A programming language used for embedded systems should support programming close to the hardware. If this is not the case, it is important that the language supports linking to code written in other languages.

In Chapter 3, we will point out some other important differences that can be of relevance to the field of real-time systems.

Table 2.1 Discrete types of the languages.

C	Java	Ada
int	int	Integer
short	short	
long	long	
	byte	
	boolean	Boolean
char		Character
wchar_t	char[3]	Wide_Character

[3]Note that Java's normal char supports Unicode.

2.9 Summary

This chapter ends in a set of factors, summarized below. In Chapter 3 we will investigate whether they are well supported by the languages or not.

Productivity

- Support for programming-in-the-large
- Automatic memory management
- Language is kept simple
- Industrial experience

Safety

- Precluding errors
- Error detection at compile-time
- Error detection at run-time
- High readability
- No ambiguities
- Language is well-defined

Concurrency

- Built in threading model
- Support for thread synchronization

Determinism

- Deterministic garbage collection
- Priorities on tasks and an appropriate scheduler
- Possibility to “wait until”

Memory footprint

- Limited memory overhead
- No oversized virtual machine

Performance

- Acceptable with very limited processor
- No interpretation in virtual machine

Portability

- Recompiling on other platforms does not change output
- Flexible enough to work without OS calls or machine code
- No implementation defined feature

Other factors

- Support for programming-in-the-small
- Programming close to the hardware
- Support for linking to other languages

3. Languages for safety critical real-time systems

In this chapter, we evaluate how well the factors identified in Chapter 2 are supported by the languages C, C++, Java and Ada. The Ada part is less extensive, as mentioned in Section 1.4. The languages are introduced by a short historical overview and the different factors are discussed under separate paragraphs.

3.1 C/C++

C came into existance in the years 1969–1973 and was initially developed in parallel with the the Unix operating system. It was developed by Dennis Ritchie at the Bell Telephone Laboratories, spread enormously and has been primarily used for system software. The language had minimalistic design goals; it was designed to provide low-level access to memory, generate only a few machine language instructions for each of its core language elements, and not require extensive run-time support. This resulted in a language suitable for many applications that had traditionally been implemented in assembly language. C is now available on a very wide range of platforms over the entire spectrum of computers, from small embedded systems to supercomputers [Rit93].

Bjarne Stroustrup developed C++ in 1983 at Bell Laboratories as an enhancement to the C language. His intention was to add support for large scale software development, and many of the new features were inspired by Simula. C++ was developed to be as compatible to C as possible, in order to enable a smooth transition. It is now a general purpose language, used both for high-level and low-level duties [Str93].

Productivity

C is a simple language [Whe07]. It does not offer much support for programming large-scale systems. It does not have support for object-oriented programming, and its capabilities for module structuring and encapsulation are weak [Bro07]. C++ on the other hand has extensive support for this. The language is however complex and cumbersome, which decreases productivity [Mic04].

There is a number of safety related problems with C and C++ (see the paragraph below). There is little error checking at run-time [Whe07], and therefore errors are detected later during development than they would otherwise be. This of course further decreases productivity.

C is the most popular programming language for real-time embedded systems [Bro07]. Another aspect worth mentioning is that many programmers have a lot of experience of the language, which supports a high level of productivity.

Safety

C and C++ are far from optimal languages from a security perspective and we clarify this by pointing out some of the imminent security flaws, listed without particular internal order:

- In C and C++ the dynamic memory management is handled manually. After

allocating needed memory, it is the responsibility of the programmer to deallocate it afterwards, when it no longer is required by the system (in C this is done with `free`, in C++ `delete` is used) [HS02].

Manual memory management introduces the risk of doing this in the wrong way. It is a common source of errors, even though it is well-known [Rob06]. Forgetting to deallocate memory results in *memory leaks*, which efficiently drain the amount of available memory. Memory leaks combined with repetition might cause the system to run out of memory. Embedded systems are particularly vulnerable to this since they often run a single program during their entire lifetime. Hence, even a very small leak can be disastrous in the long run.

The opposite, deallocating memory too early, results in so called *dangling pointers*. When the system is in a state with dangling pointers, some part of the program still uses the contents of the deallocated memory. Using corrupt memory in this way is just as disastrous as the problem with memory leaks.

There is no doubt that considerable care must be taken while managing the memory manually.

- A safe programming language should be *strongly typed*[1] and should allow few (or preferably none at all) implicit conversions between types. Explicit conversions at least show that the programmer is aware that something is going on and make them more amenable to inspection [Hat95].

  C and C++ both have some weaknesses in this area. For example, they render it possible to convert between arbitrary types via converting to void pointers [Nil06]. They allow implicit conversions from double to int, with possible unwanted loss of precision. C++ is however more strongly typed than C. In C, but not in C++, enumerations are considered identical to int [Whe07]. C allows implicit conversion from void pointers to other pointer types, but C++ does not.

- The use of pointers is almost completely uninhibited by the C language definition [Hat95], leaving much usage allowed. Dealing with pointers is however concededly error-prone. This is especially true for pointer arithmetics (for example, adding a pointer to an integer).

- The problem with dangling pointers mentioned above is not only an issue with dynamically allocated data. If a pointer `p` points to a thing `x`, and `x` goes out of scope, the pointer `p` is meaningless. Dereferencing such a pointer can lead to program failure [Hat95].

- Because of the lexical structure and syntax in C and C++, some programming errors are very easy to do. For example, it happens that the programmer types = (assignment) instead of == (comparison) or put a premature semicolon in a control structure (which can cause a totally different control flow) [Hat95].

- There are no boundary checks in arrays in C and C++, which can lead to memory corruption. C treats an array identical as a pointer, and when a pointer of the array passes the end of the array, there is no mechanism to prevent corruption of other entities [HS02].

- There are features in C and C++ which are undefined, unspecified or implementation defined [HS02]. Events with an undefined behavior means that the compiler is allowed to treat the situation as the implementor found it suitable. In C, a division by zero is an example of this. Undefined behavior is of course dangerous from a safety related point of view.

[1] The rejection of operations or function calls which attempt to disregard data types.

Unspecified behaviour concerns program constructs for which the C standard imposes no requirements and the implementor does not have to document them. Examples include the size of an int, whether a char by default is unsigned or not and more importantly, the order in which function arguments are evaluated in a function call.

Implementation defined behavior concerns features where the implementor is given the freedom to choose an appropriate approach, but has to give information to the user about what will happen. Safety related systems which must be portable, should not depend on implementation defined features [Hat95].

- Direct memory references are allowed in C and C++, jeopardizing safety [HS02].
- Being a more complex language than C, C++ adds some features which are problematic from a safety related point of view. According to Hatton [Hat95], operator overloading in C++ is allowed to such an extent that readability and understandability and therefore safety, deteriorates.
- C++ supports multiple inheritance [Str93]. This feature adds complexity and ambiguity (an example is *the diamond problem*[2]) and critics claim that single inheritance and efficient design patterns are better.

Despite the problems mentioned above, using C in safety related systems has supporters. According to Hatton, the intrinsic safety of a language is irrelevant — what really matters is how safe the use of the language can be made. If C is used with the available tool support, many common classes of errors can be detected [Hat95].

Concurrency

C and C++ have no language support for concurrency, but the functionality can be achieved using various library calls, for example POSIX[3] [Nil06].

Determinism

Neither C nor C++ have built-in garbage collection, and thus avoid this big problem to determinism. Concerning the real-time characteristics of the languages, the accessing of system time is always system dependent. In C/C++ you have to resort to external APIs, for instance the Win32 API in order to do this. The possibility to wait until a specific time depends on which solution is chosen.

Memory footprint

C does not have support for object-oriented programming, and thus does not introduce much memory overhead. C++ compilers inserts *virtual method tables* to enable *dynamic polymorphism*[4]. It also inserts temporary and anonymous objects, noticably influencing the memory needed even for smaller programs. This is however nothing significant compared to using a virtual machine.

[2] Assume that the two classes B and C inherit from A, and class D inherits from both B and C. Now a method in D calls a method defined in A that is differently overridden in B and C. Which class does it inherit via, B or C? [Ven98]

[3] Portable Operating System Interface, a family of related standards to define the application programming interface (API) for software compatible with variants of Unix [IEE07].

[4] Run-time method binding.

Performance

In an experiment by Weiskirchner, the execution times of benchmarks written in C, Java and Ada were compared. It is important to know that C does not have any checks during runtime, Java is always performing runtime-checks and in Ada runtime-checks can be switched on or off. Including runtime-checks deteriorates speed. The C-version was the fastest, but it was also the only version that did not use any runtime-checks [Wei03].

While processor speeds have increased considerably during recent years, memory access speeds have not kept pace. As a result, the cache hit ratio has a significant impact on the performance of a running program. C++ has an advantage over C in this issue, since the locality of reference is better. It is an improvement to have the class code manipulating the object data together, in an equivalent C application the data can be scattered in the memory. This is a major performance gain in C++, compensating for the performance overhead introduced by object orientation [Eve07a].

Portability

In C and C++ there are a lot of implementation defined features, for example the size of primitive data types such as int.

As previously mentioned, C and C++ have no language support for concurrency. One solution is to directly invoke RTOS services for concurrency management, but that constrains the program to platforms that run that RTOS. Another possibility is to use a POSIX binding, but this still leaves quite a bit of functionality implementation-dependent [Bro07].

Other factors

C and C++ both support programming close to the hardware. Using pointers, it is possible to write to arbitrary parts of the memory.

3.2 Java

Java was developed by Sun Microsystems. The language derives much of its syntax from C and C++, but is in many aspects different from them. Java was designed to be a safe and platform independent language. Platform independence is achieved by compiling Java code to byte code, which then can be run on a virtual machine.

There are some features in Java which deteriorates the deterministic behaviour. To be able to use Java in real-time systems, the Real-Time Specification for Java (RTSJ) has been developed. By introducing new memory areas, it solves Java's problem with non-deterministic garbage collection [Mic04]. Mackinac is the name of Sun's implementation of the specification [Mic04]. Another implementation, JamaicaVM from Aicas, relaxes some of the restrictions of the specification [Aic07].

An alternative to using a virtual machine, is to compile the Java code to native code. This is the intention of the Java-to-C compiler LJRT (Lund Java-based Real Time), developed at the Department of Computer Science at Lund Institute of Technology [Nil06].

Productivity

Java has support for object-oriented programming and thus has support for classes, inheritance and virtual methods. Industrial experiences have showed that object-oriented techniques are better suited for structuring code than the alternatives [Nil06]. The fact that Java is slightly sparse in its features favors productivity. For example, it does not allow programmer defined operator overloading [NP97].

Using RTSJ is probably not as productive as using Standard Java. One reason for this is that new memory areas are introduced, *immortal* and *scoped memory*. These areas are not garbage collected, as described in the determinism paragraph. The new memory areas increase the complexity and lower the simplicity of Java [Ive03]. This is of course only true for implementations that strictly follow the specification. In JamaicaVM, the limitations that RTSJ imposes on the use of memory areas are relaxed [Aic07].

In LJRT, memory is managed in the same way as in Standard Java. Therefore, using LJRT should in theory be as productive as using Standard Java and more productive than using RTSJ. However, the LJRT project is a research prototype, and not yet of commercial quality. Considerable configuration work is now needed to get the wanted output from the system, since the prototype is especially constructed to be extensively configurable.

Safety

Java is a much safer language than C/C++, which we show by a direct comparison to the problems in paragraph safety in Section 3.1. Below are the safety related aspects previously discussed for C/C++ investigated for the Java case:

- Java uses automatic memory management, which means the system itself handles object deallocation through the garbage collector. The programmer cannot cause memory leaks and dangling pointers, even by poorly written code.

  To a lesser extent, this is also true for RTSJ, but the use of immortal memory is quite dangerous, since objects allocated there will not be reclaimed automatically. Memory leaks can result from improper use [Aic07].
- Java allows fewer implicit conversions than C/C++. For example, Java does not allow implicit conversions from double to int [GJSB05].
- Java does not have pointers, but references. A reference in Java is a safe kind of pointer. The only way a reference can be manipulated is by assigning it to an object. This means that you cannot do pointer arithmetics with references [NP97].
- In Java, objects cannot be allocated on the stack, and references cannot point to primitive datatypes[5]. This means that a reference cannot point to anything that has gone out of scope [GJSB05].
- Java uses C-based lexical structure and syntax, and therefore has the same problems in this area (for example, = vs. ==) [GJSB05].
- Every time an element in an array is accessed, there is a run-time check to see that it is within the array bounds [GJSB05].

[5] Primitive datatypes (for example characters and integers) are data types provided by a programming language as basic building blocks.

- Java is a more well-defined language than C and C++; it has less undefined, unspecified and implementation defined features. An important reason for this is that Java abstracts away from platform specific behaviour [Eck07].
- Direct memory access is not possible [GJSB05].
- In contrast to C++, Java does not support operator overloading [NP97]. According to Hatton, operator overloading undermines security [Hat95].
- Java does not support multiple inheritance [NP97].

Concurrency

Java has a built-in threading model and synchronization primitives. Synchronization can be achieved by letting each synchronized Java object constitute one monitor [Nil06].

Determinism

To be able to use Java in hard real-time systems, the lack of determinism must be attended to. One issue weakening the determinism of Java is that classes are loaded dynamically when needed. This results in an uncertainty of whether a class already is loaded or not, when the matching object is allocated. If it has not been loaded, it has to be done dynamically at that time. The problem can be solved by loading all classes when the execution begins [NEN02].

An important reason why Standard Java cannot be used for real-time systems is the automatic garbage collection of the language. The problem is that the garbage collector causes unpredictable latencies on other threads [Nil06].

In Java, it is possible for threads to indefinitely block higher-priority threads since no *priority inheritance*[6] is implemented. This is not acceptable for real-time systems. Another problem with Standard Java is that it lacks a "wait until" method [Årz03].

RTSJ solves the problem with determinism of Standard Java by introducing scoped memory and immortal memory. These certain memory areas are intended for hard real-time threads, and the garbage collector never collects any objects there. In scoped memory, all objects are destroyed when the application goes out of scope, and in immortal memory the objects are never destroyed. RTSJ also supplies a normal heap for soft real-time threads, and automatic garbage collection is used there [Mic04].

The classes of RTSJ offers functionality to wait until a specific time. According to this specification the class `HighResolutionTime` used for this purpose has a nanosecond granularity.

JamaicaVM uses deterministic garbage collection, and therefore any thread can use the normal garbage collected heap. This relieves the developers of the burden of complex memory management. Nevertheless, the new memory areas can be used if the application developer wishes to do so [Aic07].

The memory management in LJRT is the same as in Standard Java, but deterministic garbage collection is used. Deterministic garbage collection has been proven possible by Henriksson [Hen98]. His idea was that a separate GC thread should run when no threads of high priority were running. When this thread was called, it was

[6] When a task blocks one or more high priority tasks, it ignores its original priority assignment and executes its critical section at the highest priority level of all the tasks it blocks.

supposed to conduct an amount of GC work, proportional to the amount of memory allocated by high-priority threads [Rob06].

The issue about waiting until a specific time is solved in LJRT; it is supported in the class `RTThread` by the `sleepUntil` primitive in the LJRT library [Pro07].

Memory footprint

A Standard Java virtual machine has a memory footprint of tens of MBs and is rarely applicable for embedded systems because of their constrained environments. To run Java on embedded systems, Java Micro Edition has been developed by Sun Microsystems. It has a very small virtual machine, but also has significant limitations [NEN02].

Mackinac, Sun's implementation of RTSJ, uses a virtual machine and has the same problems as Standard Java when it comes to memory footprint. The solution uses Initialization-Time-Compilation (ITC), which means that the code is compiled at program start. This increases the amount of required memory further [Rob06]. There is however no explicit reasons that RTSJ could not be used together with smaller virtual machines using less memory.

JamaicaVM occupies about 256 kB of memory. Small applications that make limited use of the standard libraries typically fit into about 1.5 MB of memory [Aic07].

As previously mentioned, LJRT uses no virtual machine at all. By compiling (via C) to native code, the memory footprint can be kept very small. [NEN02].

Performance

There are several factors making programs written in Java slower than equivalent programs implemented in C/C++. The fact that you cannot allocate objects on the stack, but are restricted to using the heap contributes to making the language slower than C++ [Eck02]. The most significant performance loss comes however if the Java byte code is interpreted in a virtual machine. It is then only about 10 % of a comparable C++ program [NEN02].

To improve performance and make it execute faster, Just-in-Time compilers (JITs) are used. Their approach is to during execution compile byte code to native code, which for the currently best compilers results in code approximately as fast as C++ code. There are however two main disadvantages with JITs. They require much memory and they are difficult to combine with requirements of high level determinism [NEN02]. Therefore, it is hard to use Standard Java for real-time systems and retain speed.

Mackinac uses, as mentioned ITC. Compared to normal interpreting of code, the speed is increased. Contrary to JIT, ITC causes no problems to the deterministic behaviour [Mic04].

JamaicaVM lets developers compile portions of Java applications into machine code, using C as an intermediate language [Aic07]. Code that does not execute frequently, for example startup and error handling code, can be left in Java byte code.

In the experiment by Weiskirchner mentioned in Section 3.1, the difference in execution time between Jamaica and Ada was less than between different Ada runtime environments as long as runtime checks were used. C was faster than Jamaica, but while C does not have any checks during runtime, Java is always performing runtime-checks [Wei03].

Native code is obviously much faster than interpreted code running in a virtual machine. Experiments have shown that the same applies for the native code generated by LJRT. The LJRT code is however slower than handwritten optimized C code and more controversially, slower than a Java virtual machine with Just-in-Time compilation. The reason for this lower performance is the GC synchronization of the compiled code, giving longer execution times. Through various optimizations of LJRT, this could to some extent be accounted for. Without the synchronization, the performance of LJRT would be much closer to good handwritten C code, but would then only be applicable in applications with exclusively static memory allocation[7] [Nil06].

Portablity

Java was created with portability in mind. The idea is that you should be able to write and compile the program once and then run everywhere. Of course, this requires that the virtual machine is available (and correctly implemented) everywhere.

Java abstracts away from platform specific behaviour, and thus tries to have as few implementation dependent features as possible [Eck07]. Because Java has a large standard library with for example support for concurrency, there is no need to call external code to achieve this functionality.

LJRT translates Java code to C code. Because of for example system calls, the generated C code depends on the target platform. This means that the Java-to-C compiler has to support various platforms.

The code resulting from LJRT has been executed on various processors ranging from small 8-bit CPUs to high-end embedded CPUs, and therefore the portability of LJRT seems to be good [Nil06].

Other factors

Direct memory access is not possible in Standard Java. Programmers can work around this problem by using Java Native Interface (JNI), a framework that allows Java code to call and be called by native applications written in other languages. This does not change the fact that support for programming close to the hardware in Java itself is poor.

RTSJ uses a so called `PhysicalMemoryManager` for physical memory access [Mic04].

We have seen that compiling Java code to native code has some advantages. Unfortunately, it also has at least one disadvantage: it negates Java's write-once, run-everywhere approach.

The main advantages of, like LJRT, using C as an intermediate language are [NEN02]:

- Since there already exist well-tried compilers for most CPUs, there is no need to support multiple back-ends.
- It gets easier to link the application to external code, for instance required device drivers.

[7] Static memory allocation refers to the process of allocating memory at compile-time before the associated program is executed.

However, there is a danger in using C as an intermediate language. It is possible, and according to Hatton [Hat95] not at all uncommon, that the compiler could generate features of C known to to be problematic (for example implementation dependent), and the original source code developer will be unaware of this.

3.3 Ada

This section briefly describes some relevant characteristics of Ada. It does not reach the same level of impartialness as the other languages and solutions in the study due to lack of time. Most sources used here are subjective. There certainly exist eager advocates and lobby organizations. Consequently, the reader should be somewhat more sceptical while looking through this section. Still, the language inarguably contains interesting aspects worth mentioning and therefore deserves its part in the study. The critics of Ada, however, claim that it is a large and complex designed-by-committee language.

In the beginning of the 1970s, the prices on hardware fell and the American Department of Defense started to switch focus to the prices on embedded software. The department conducted a survey in 1973 that showed that three billion dollars were spent on software and at least 450 programming languages and incompatible dialects were used in their embedded systems [BW01]. In 1976 they carried out an evaluation of the existing languages to formulate requirements for the desirable goal, a single language. The requirements led to the birth of Ada in 1983.

Ada 95 was published in 1995 as an update to reflect new programming language standards and support for object-orientation was added [Årz03]. Ada has been mostly used in embedded systems for military and aerospace purposes, but the Ada community has strong and active advocates, pushing for extended industrial use. In 2005, Ada 2005 was released. This version included new scheduling policies, new timing events and standardized the *Ravenscar profile*[8] [Rog06].

Productivity

Ada supports programming-in-the-large, i.e. contains language features that support decomposition and abstraction [Årz03]. The latest versions have support for object-oriented programming and have explicit support for modules in the form of packages, declared in two parts: a *specification* and a *body*. The packages enable encapsulation, since only the entities declared in the specification are externally visible. *Separate compilation*[9] of modules is integrated into the Ada language specification. It is also more reliable than linking together precompiled units as it is done in C [BW01].

Ada is a powerful language and it contains some of the features that makes C++ cumbersome and complex, like operator overloading and generic programming. It is not a simple language, but strict and safe. This creates a steeper learning curve, effecting the productivity.

In a study by McCormic it is claimed that the use of Ada, compared to C, significantly improves the ability to complete projects on time. McCormic's experience is based on multiple groups of students building software to satisfy identical requirements. Some of the groups used C to implement their systems while other used Ada [Con07].

[8] Subset of the Ada tasking features designed especially for safety critical hard real-time systems.

[9] The entire program does not need to be recompiled after each minor change.

Safety

Safety was very much in focus when Ada was designed. First of all it is a strongly typed language. The language defines numerous static and runtime checks for unsafe conditions such as integer overflow and dereferencing invalid pointers [Dun07]. Other features that can be brought up are that Ada does not require a custom copy constructor to achieve *deep copies* of objects, macros are not permitted and that void pointers are absent. Multiple inheritance is also not present as a language construct.

Some weaknesses of the language can however be identified. Ada was designed to permit, but not require, automatic memory management. If the garbage collector is absent, the programmer is responsible for the memory management and well-known risks arise [BW06]. The problems are however somewhat reduced by the fact that the natural storrage of Ada is the stack and not the heap. This means most objects are created on the stack, which simplifies the memory management. One further weakness of Ada is that it supports operator overloading, which negatively influences the readability of the language.

In the study by McCormic mentioned earlier, he claims that Ada not only increased productivity, but also dramatically reduced the number of defects in the students' programs [Con07].

Concurrency

Ada has low level synchronization primitives, which can be used to implement semaphores. A higher level synchronization construct, *the protected object*, is also included. It is a form of monitor, which that allows access to internal data only through protected functions. Those protected objects are however not fully integrated in the object oriented model and they cannot be extended further [BW01]. Synchronization and communication using message passing is included in the language and it is also possible to dangerously use global variables for communication.

Ada 2005 provides various scheduling policies, such as non-preemption within priorities, round robin using time slicing, and Earliest Deadline First [RD06].

Determinism

The language semantics of Ada are generally well-defined, but there are features whose effects are implementation defined or unspecified [BW06], thus making static code analysis harder. Access to a real-time clock is provided through the packages called `Calendar` and `Real_Time`, and a granularity of a millisecond is available. Some parts of the packages have implementation dependent accuracy and range. Absolute time delays are a natural part of the language.

Ada without garbage collection avoids the possible problems involved and to further increase the determinism of Ada, it is possible to use the somewhat restrictive *Ravenscar* profile mentioned earlier.

Performance

In the experiment by Weiskirchner previously mentioned, the performance of two Ada compilers was compared with C and Java. Switching runtime-checks off in Ada, the version compiled with the Ada-GNAT [Fre07] compiler was faster than the C-version, while the version compiled with Ada-Multi [Gre07] was not [Wei03].

Portability

There exist implementation defined features in the language, reducing the portability of Ada applications. The interface to any given platform is however defined in the package `System`. The fact that Ada is an internationally standardized software language, and the fact that all Ada compilers have to be validated result in highly portable software.

Other factors

Ada supports linking to code written in other languages and insertion of native code is possible [RD06].

For control systems it is rarely enough to satisfy the computational needs with integer arithmetic, they need to make use of real numbers. Ada has better support for working with real numbers than our other candidates. There are mainly two ways of representing real values: *floating-point* or *scaled integer*. Floating-point numbers are an approximation of real numbers and always have a relative error, the difference to the corresponding factual value. Scaled integers is a product of an integer and a scale.

Commonly, programming languages support a floating-point type for an implementation dependent precision. Ada has this and in addition provides constructs to create floating-point numbers with different precision and fixed-point numbers, implemented as scaled integers. To construct a certain fixed-point the programmer needs to specify the range within it will lie and an absolute error bound [BW01]. Neither C/C++ nor Java can offer this simplicity.

3.4 Summary and discussion

Although C and C++ have excellent performance and low memory footprint, the languages also have several drawbacks. C offers little support for programming-large-scale systems, making it an unproductive alternative. Although many programmers have a lot of experience of the language, this fact alone cannot compensate for the productivity drawbacks C has. Having support for object-oriented programming, C++ is much better than C for programming large-scale systems, but the complexity of the language is a problem.

C and C++ have enormous problems in the safety area. Among other things, it can be mentioned that the dynamic memory management in these languages is handled manually, that the languages allow several implicit conversions and that the use of pointers, which is error-prone, is almost completely uninhibited by the C language definition. Using the available tool support, many errors can be detected, but this requires that the tools are correctly used. We believe that this is an unfavorable extra step, especially since there exist languages which have great intrinsic safety.

The problems in safety and productivity that C and C++ have should alone be enough to consider other languages. As we have shown, Java shines in theses areas and it also has great portability. Unfortunately, because of the using of automatic garbage collection, Standard Java lacks determinism and therefore cannot be used in hard real-time systems. RTSJ and LJRT solve this problem either by introducing new memory areas or by using deterministic garbage collection.

Mackinac and Jamaica, the implementations of RTSJ we have mentioned, both

have acceptable performance, and the same is true about LJRT. Contrary to RTSJ, LJRT compiles Java code to native code, which means a lower memory footprint. Another advantage with using LJRT is that, because it uses C as an intermediate language, it gets easier to link the application to external code. These advantages are important and the reason why we have chosen to use LJRT for 800xA. Admittedly, there exist dangers in using C as an intermediate language, but we believe that the advantages are greater.

Ada was the last language in our study, a language with a problem oriented design. During the development of the language, many contemporary solutions were reviewed, and the intention was to make Ada the superior choice for real-time software. The result, also shaped by two updates of the language, is rather complex and verbose. On the other hand, the language promises great safety and very good concurrency and real-time features, combined with support for developing large scale systems. Whether it is going to be as widespread as the promoters work for, is left to be seen.

Tables 3.1 and 3.2 summarize how well the different alternatives we have discussed fulfill the requirements on programming languages for embedded real-time systems. The scores should not be taken as objective facts, but rather as our well-motivated opinions.

Table 3.1 Characteristics of different solutions - part I

	C	**C++**	**Ada**
Productivity	Bad	Doubtful	Doubtful
Safety	Bad	Bad	Good
Concurrency	Bad	Bad	Good
Determinism	Doubtful	Doubtful	Good
Memory footprint	Good	Good	Good
Performance	Good	Good	Good
Portability	Bad	Bad	Good
Other factors	Good	Good	Good

Table 3.2 Characteristics of different solutions - part II

	Standard Java	**RTSJ**[10]	**LJRT**[11]
Productivity	Good	Doubtful	Good
Safety	Good	Good	Good
Concurrency	Good	Good	Good
Determinism	Bad	Good	Good
Memory footprint	Bad	Doubtful	Good
Performance	Bad	Good	Doubtful
Portability	Good	Good	Good
Other factors	Bad	Good	Good

[10] A specification.

[11] A Java-to-C solution.

4. Description of 800xA

In this chapter we give a description of 800xA, including the redundancy functionality and the component Role Selection.

4.1 Overview

The control system 800xA is a solution that integrates advanced control functions, production management, maintenance management, information management and network management. It is intended for industrial use within a wide variety of applications where a certain focus on safety and reliability is required, from oil platforms to power plants.

The basic parts of the 800xA system are computers, sensors, actuators, communication units and software with program modules. The system is connected to sensors, from where it receives input signals, for example values of temperature and flow. 800xA collects the information in a process station. The station has control programs that receive the signals, compare the values with some reference values, and send out control signals to actuators at various positions.

A single module of the hardware we worked with is shown in Figure 4.1. The figure shows the diagnostic LEDs indicating the current state of the processing module (PM) and its current *role*, as described in Section 4.3. Under the LEDs, the only button (used for resetting the PM) and the slot for the flash card (which we used to load the software) is located. The other visible components in the figure are the backup battery (needed in the case of power failures) and different connectors for power and communication.

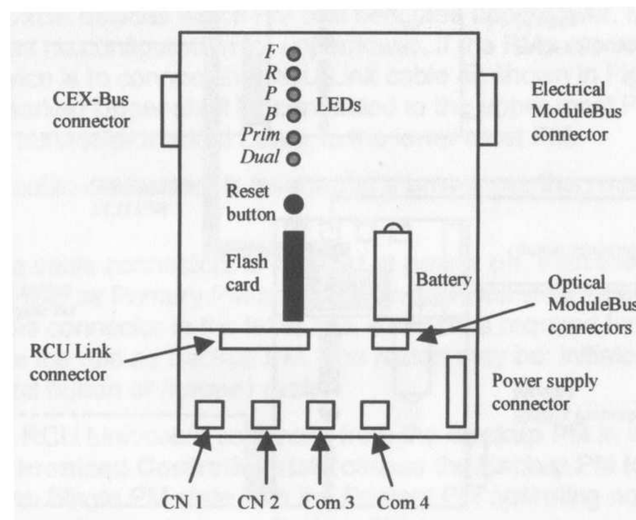

Figure 4.1 Schematic figure of a single PM.

The 800xA software is divided into four main parts:

- **Control Builder**. Lets the user edit, compile and test the wanted controller program and finally download it to the Controller.

- **Controller**. Executes the program and configuration developed in the Control Builder.
- **Control Aspect System**. Handles operations introduced by other so called aspect systems.
- **OPC Server**. Also used for communication with other aspect systems.

Our work regards the controller, which is further divided into subsystems. The component we have ported to Java is called *Role Selection*, a part of the subsystem providing the redundancy functionality of the control system 800xA. The redundancy is presented in Section 4.2 and the module Role Selection in Section 4.3.

4.2 Redundancy

The purpose of the redundancy functionality is to increase the availability and reliability of the controller, by having a secondary module in the system. This backup module acts as a so called "hot standby" and can take the role as primary module in case of a hardware error. The user application does not need to be designed for use with a dual system and does not need to be aware of this option. Figure 4.2 shows a typical configuration of two modules, connected to enable synchronization of memory and I/O. The direction of the cables decides the *geographical positions* of the PMs, "upper" or "lower".

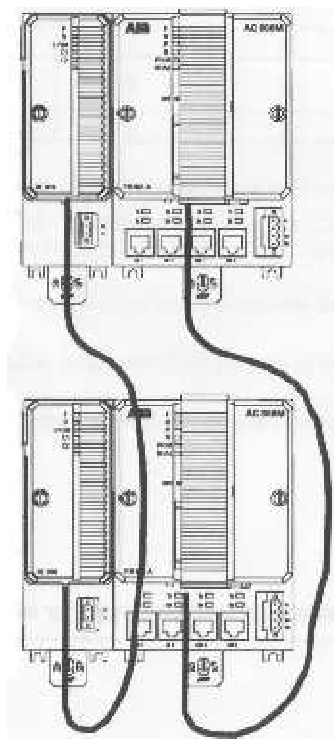

Figure 4.2 Schematic figure of a two connected PMs.

The actual takeover is a bumpless[1] transfer, since it is ensured that the outputs coincide at the time of the switching [Hel06]. This is guaranteed by a duplication

[1]Smooth transfer of a controller from one operating mode to another.

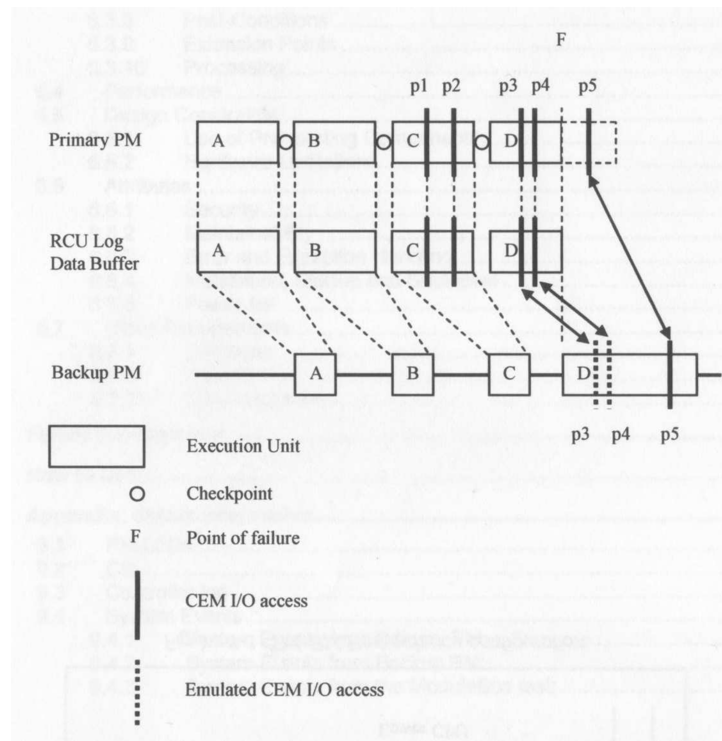

Figure 4.3 Example of takeover following a failure in the forth execution unit.

principle, which regularly updates the backup unit to put it in the same state as the primary, currently running unit (see Figure 4.3).

The duplication principle is implemented by dividing the execution of the user application into execution units. During execution, the contents of the primary memory and registers are buffered. After each execution unit there is a checkpoint, and the buffered data is copied to the backup unit. If a hardware error is detected in the primary unit, the backup unit can resume execution from the latest point a content transfer took place (the latest checkpoint).

Since the execution takes over from a time in the past, it is important that output signals are not repeated. To make certain this can be avoided, all I/O accesses are logged and buffered in the RCU Log Data Buffer. When there is a takeover and the execution is restarted from the latest checkpoint, the buffered signals are only emulated.

4.3 Role Selection

The component Role Selection is a part of the redundancy functionality in the control system. It is used every time a controller with redundancy is started or re-started and assigns the role of primary unit to the more suitable of the two candidates, while the other one becomes the backup. The module offers methods to get and set this role and to subscribe to changes.

The role is selected by evaluating a characteristic called the *eagerness*. This value is calculated according to a number of parameters and the lower the resulting value is, the more eager the unit will be to become primary.

The eagerness is calculated and compared without any communication between

the two units; instead they compete about becoming the primary. When both units have indicated that they are alive, they both try to connect to the so called ABB Communications Expansion Module (CEM). The time the unit waits before trying to connect is the eagerness value multiplied by a constant. The one that is first will connect and become the primary unit; the loser will fail because the CEM bus already has a connection. The procedure will be explained further in Section 5.1.

5. Introducing Java in 800xA

This chapter describes the practical work involved in implementing a module of 800xA in Java. It also presents our solution for making it work together with the rest of the system and some adaptations we had to make.

LJRT had earlier mainly been tested on hosts running on GNU/Linux or Solaris. Since our work stations were running Windows, the solution we had to conform with was to use Cygwin [Red07] to simplify the porting. This variation in development environment meant that some modifications were needed before even minimal Java examples could be compiled from Java to C. Parallel development of LJRT during this stage solved the problems and they should not appear again; two of the problems were that Windows and Unix uses different path separators and that Windows paths are not case sensitive.

When a working experimental setup was available, we were able to produce C code from our Java. The idea was then to include the generated files in the current ABB development solution, Microsoft Visual Studio 7.1, and remove the old implementations. Again adaptations were needed to make it compile and link, both in the files generated by LJRT, the existing 800xA code and the project configuration of the development environment. A full description of our changes is included in Appendix A.

As a second step, to test the thread functionality of Java, we have implemented a thread monitoring the geographical position of the both modules, reporting when they change. The two possible geographical positions are “upper” and “lower”, determined by the cable connection between the modules. This thread in turn starts another thread printing the current MAC and IP addresses of the both modules. To make this work in the environment of 800xA, numerous changes needed to be done. The work is thoroughly described in Section 5.2.

5.1 Porting the module Role Selection

The existing C/C++ solution of Role Selection consisted of three classes:

- **RoleSelection**. The most significant of the classes. A static class that among other things included an enumerated type for the different roles of the CPU, functions called from the rest of the system to get and set the role, and a function to assign a CPU role at start-up. It also had a function to let other functions subscribe to the changes of the CPU role, achieved using function pointers.
- **CCpuRoleHandler**. This class provided an interface to manage the role selection. The functions set, get and subscribe were requested from this class. CCpuRoleHandler was implemented using the singleton design pattern.
- **NonVolatileMemory**. A smaller static class, containing functionality to set up the base address where attributes that shall survive power failure should be stored. Functions for reading from and writing to this battery backuped memory were also provided and an enumerated type with identities for the non-volatile attributes.

Our Java implementation is presented in the following paragraph. Some design solutions are explained in paragraph “Design on class level” and encountered issues we had to deal with are described in Section 5.3.

Class internal implementation

The porting to Java was pretty straight-forward. However, some things had to be done differently than in the original implementation.

- The original implementation used enumerated types. For example, the enum `CpuRole` defined the various roles the CPU could have (primary, backup, undefined, etc.). Our implementation of LJRT used Java 1.4, a version that lacks support for enums. As a result, our Java implementation uses inner classes with static constants instead.

  This is the original C code:

  ```
  enum CpuRole {
      CpuRoleUndefined = 0,
      CpuPrimary = 1,
      CpuBackup = 2
  };
  ```

  This is our Java solution:

  ```
  public class CpuRole {
      public static final int CpuRoleUndefined = 0;
      public static final int CpuPrimary = 1;
      public static final int CpuBackup = 2;
  }
  ```

  This solution is not ideal. The good thing is that, when reading values, you can use static constants instead of “magic numbers”. The downside is that you cannot declare variables of types like CpuRole and therefore you have to use the primitive data type `int` instead. Naturally, you could create an object of the inner class, but this solution would be ineffective.

- If you want more than one return value from a C function, you can give it pointers as arguments and manipulate at the corresponding addresses. In Java, you could instead pass an object as argument and store the second return value there. We chose a more effective solution. We stored the return value in a variable local to the object whose function we called, and then we fetched the variable with a separate function.

  Consider the C code below:

  ```
  int ret_value1, ret_value2;
  ret_value1 = read_data(&ret_value2);
  ```

  Assume that ret_value2 is assigned a value in the function read_data in the example above. In Java, the functionality was achieved in the following way (using a static class):

  ```
  int retValue1, retValue2;
  retValue1 = MemClass.readData();
  retValue2 = MemClass.getRetValue2();
  ```

- The original implementation stored pointers to functions in an array, and called the functions that the pointers pointed to. Neither of this is possible in Java.

  Our solution was to convert the pointers to integers before calling the Java module. Then the integers were stored in a Java array. There is no way to call the corresponding functions from Java, so this had to be done from native code.

  The code below shows our solution. In a C++ file, a pointer is converted to an integer:

```
void subscribe(void (*functionToCall)())
{
    int param = reinterpret_cast<int>(functionToCall);
    // Now we call the Java method subscribeCpuRole with
    // param as argument
}
```

  In a Java file, the following code is present:

```
private int[] subscribers;   // Initialized with zeros

public void subscribeCpuRole(int functionToCall) {
    int i = 0;
    while (subscribers[i] != 0 && i < subscribers.length)
        i++;
    if (i < subscribers.length)
        subscribers[i] = functionToCall;
}
```

  When calling the functions the pointers are pointing at, the following Java code is executed:

```
int i = 0;
while (subscribers[i] != 0 && i < subscribers.length) {
    nat_callFunction(subscribers[i]);
    i++;
}
```

  In a C++ file, we have this code:

```
void nat_callFunction(int func)
{
    void (*function_to_call)() =
    reinterpret_cast<void (*)()>(func);
    function_to_call();
}
```

- The module is reading from and writing to specific addresses of the non-volatile memory. These things cannot be done directly from Java, so native code had to be used again.
- Procedures in the Java module have to be called from the rest of the system, and the Java module has to call procedures in the rest of the system. To make these things possible, we wrote two bridges for all the interlingual procedure

calls to pass. Our Java code uses the C file `frommodule.c`, and its procedures are declared in the Java code and marked as native. The bridge solution is described further below.

The Java code we developed for the module Role Selection was somewhat simple, static and singleton classes do not put much demand on the garbage collector and our code did not contain any thread functionality. To make up for this we added some functionality in new Java threads, repeatedly created objects, and let the garbage collector take care of the dead objects. This work is presented in Section 5.2.

Design on class level

LJRT generates C code from Java code. However, our module sometimes needs to call C++-functions in the existing implementation of the system. To enable those interlingual calls, we developed a bridge solution consisting of three files.

- Our calls from Java went from our source code, to C code complying with the syntax and name mangling required by LJRT named *frommodule.c*, to a C++ file name *frommodule2.cpp*, and then to the desired function of the 800xA system, see Figure 5.1. The C++ file enables calls to member functions and is not always needed, nevertheless we let all calls consistently go all the way over the bridge. A short example of what this might look like can be seen below:

  1. In a Java class Example, the following function is called:

     ```
     private native static int getValue(int position);
     ```

  2. In the C file the following code is present:

     ```
     extern int bridge_getVariable(int position);

     GC_VAR_FUNC_BEGIN(int, Example_getValue_int,
     int position)
     GC_FUNC_ENTER
        GC_RETURN_VAR(bridge_getValue(position));
     GC_FUNC_LEAVE
     GC_VAR_FUNC_END(int, Example_getValue_int)
     ```

  3. In the C++ file we have this code:

     ```
     extern "C" {
        int bridge_getValue(int position);
     }

     int bridge_getValue(int position)
     {
        return someFunctionCall(position);
     }
     ```

- The calls from the existing system all went over another so called bridge, implemented as *tomodule.cpp* (see Figure 5.1) with the corresponding declarations in the header file *tomodule.h*. All occurrences of inclusion of headers from the original Role Selection files were changed to this, in order to streamline the ingoing procedure calls. Below follows an example implementation with its needed LJRT syntax, see also Figure 5.2.

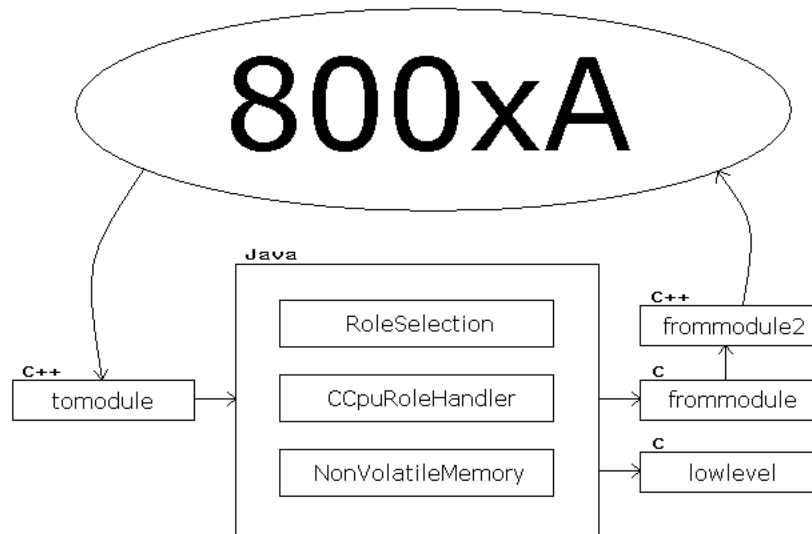

Figure 5.1 Our solution using a bridge for calls to and from the 800xA source code.

1. Assume that somewhere in the existing 800xA code, the following two lines are present:

```
#include <tomodule.h>
bool status = cpuRoleSet(2);
```

2. In *tomodule.cpp* we have this code:

```
bool cpuRoleSet(int cpuRole)
{
   int param = cpuRole;
   bool ret_value;
   GC_VAR_FUNC_CALL(ret_value,
   RoleSelection_cpuRoleSet_int, param);
   return ret_value;
}
```

3. Finally the procedure called in the by LJRT generated C code might look like:

```
GC_VAR_FUNC_BEGIN(bool,
RoleSelection_CpuRoleSet_int, int crCpuRole)
   bool status;
   GC_FUNC_ENTER;
   /* Some code setting the role and assigning a value
   to status */
   GC_RETURN_VAR(status);
   GC_FUNC_LEAVE;
GC_VAR_FUNC_END(bool, RoleSelection_CpuRoleSet_int)
```

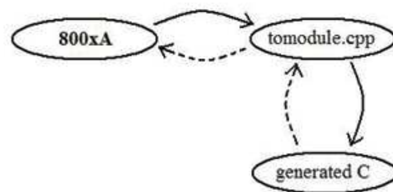

Figure 5.2 Calls from the 800xA system to our generated C code. The dashed lines represent returns from calls.

5.2 Including Java threads

As previously mentioned, we wanted to include more of the possibilities and strengths of Java in this work. We implemented two Java classes extending the Thread class and added functionality enabling us to observe if the behavior was correct (see Figure 5.2).

- **RoleChangeSupervisor.java**. This is a thread we start in both the primary and backup module at the same time as many other diagnostic threads are started in 800xA. It contains an infinite loop checking whether the geographical position of the module has changed every three seconds. Each loop iteration it also creates an *MACIPPrinter* object and starts the thread described below. When the system has reached the state when a switchover is possible (the both PMs are synchronized), it waits another seven loop iterations and then forces the primary module to shut down. This thread is started from C++ code.
- **MACIPPrinter.java**. This is a minor thread, calling a native function once to print the MAC and IP addresses of the current module. It is always terminated shortly after its creation to test the garbage collector. These threads are started from Java code. The IP address should always follow the primary module and that this was working properly during takeovers could be confirmed.

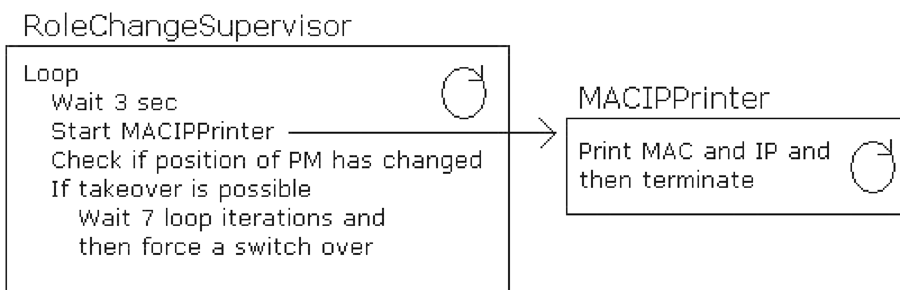

Figure 5.3 The two Java threads we run in the system.

ABB uses VxWorks, and LJRT had no implementation of Java-threads using VxWorks system calls. The best thread implementation LJRT offers, is *Pthread*[1], but this did not constitute a viable option.

We had to translate the Pthread calls in LJRT so that VxWorks could understand them. Instead of directly using VxWorks system calls, we chose to use an existing solution at ABB. Threads are handled through an ABB internal interface called *Virtual*

[1] A POSIX standard for threads.

Operating System (VOS). This required changes in the thread model used by LJRT. We had to map the existing Pthread calls to corresponding VOS calls. We also needed to make some changes in the implementation of the LJRT incremental garbage collector, which was used during this project. Those changes are documented in Appendix A.2.

5.3 Difficulties encountered and lessons learnt

While working with this project we encountered a number of problems at different stages. The work involved in single stepping through the code with a hardware debugger to find the causes was very time consuming.The problems of highest interest for other developers are presented in this section and can hopefully simplify possible future work concerning this domain. The issues are presented in the order we encountered them.

- By default, *dead code elimination*[2] is enabled in LJRT. Initially, not much of our Java code was compiled to C. Disabling this optimization was an option, but it resulted in generated code of immense size. By including a run-method in all classes (even though they have nothing to do with threads) systematically calling every internal method, all code is generated and the resulting overhead is small. Also, the run-methods can be removed afterwards.
- In some files written in C++, we include header files written in C. Some of those header files use the user defined types *class* and *this*. Those words are reserved words in C++. We solved this problem by having this code in the C++ files:

```
extern "C" {
   #define class class_t
   #define this this_t
   #include <somefile.h>
   #undef this
   #undef class
}
```

- The code snippet for initializing the Java classes (further described in Appendix A.1) could not be placed too early in the startup sequence of 800xA. Our first attempts to do this were a bit too eager and the later following initializations of global variables in 800xA efficiently destroyed our Java classes.
- The default heap size stated in the configuration file of LJRT is too large. This showed by 800xA failing to bind IP addresses in the backup module and led to problems and crashes during execution. Another heap related problem was encountered when we had introduced the threads. With threads enabled, the Java classes needed more heap space for its initializations and we happened to allocate too little. As a result, our inner classes could not be correctly created, resulting in system crashes.
- To ensure that the execution of 800xA runs as it should, it is required that a so called *software watchdog* is called repeatedly. The watchdog had a function

[2] A compiler optimization used to reduce program size by removing code which does not affect the program.

called *kick*, which should be called to prevent the triggering of a stall timer. Initially, our lack of this knowledge led to a system stalling from time to time during our development work.

- When we were testing our implementation of Java threads, we once again worked too early in the startup sequence of 800xA. It is of course necessary that the RTOS, VxWorks in this case, is properly started before it is used to spawn any new threads.

6. Results

In this chapter we describe the behaviour of the system we have observed during our development work. First of all, the final version of our solution appeared to operate as it should. Apart from claiming that we had a naïve feeling of a properly working system, we will present three factors confirming the correctness of our Java solution and its possible effects on the rest of the 800xA system. The first test described below, a formal ABB test suite, was conducted after the completion of the porting of Role Selection system. The two following tests were performed throughout the development, since they were integrated in the development cycle during debugging and normal execution. The last two tests containing measurements of performance and memory footprint were performed for the version including our Java threads.

1. **Function Type Test Description PM Redundancy** [Ang07]
   This document describes the functional testing of the redundancy functionality, and was used as a part of the standard regression testing. The test cases described required manual work and included both normal action and alternatives resulting in failures. We performed a subset of those test cases[1], including testing of: Initialization, hardware replacement, IP address assignment, removal of connections, and power failures. Some test cases were omitted due to limitations in the experimental setup. All completed tests ended with approving results, which is our main claim that the system, with our integrated Java module, operates correctly.

2. **Testing the garbage collector**
   Our final version of the system has never shown any signs of problems with the automatic memory management. We have tested creation of thousands of objects in infinite loops without encountering memory problems. As described in Section 5.2, our system continually creates new threads and everything executes smoothly. We have observed the behaviour of an LJRT internal variable, representing the heap pointer of the LJRT heap. It increases when new objects are allocated and, when there is no memory left, it decreases, which means that the garbage collector does its work. This also means that the GC is of type "stop the world" — a non-deterministic type of GC. LJRT supports an incremental, deterministic GC, but we did not fully implement this functionality and leave it as future work. We discuss this further in Section 7.1.

3. **The performance and the stall timer**
   As mentioned in Section 5.3, the 800xA uses a so called *software watchdog* during the startup sequence together with a stall timer, set to 2 ms for the release version of the system. This watchdog needs to be called frequently, while executing sections where interrupts are disabled, to ensure that the system runs properly along the correct execution path. The startup sequence is such a section. This path contains calls to the `kick()` function of the watchdog at certain points in the source code, ensuring that the expected control flow is followed. Our final version of the Java code does not contain any new calls to the watchdog, and the system does not stall. This indicates that the executions time of the Java module does not exceed the upper bound of the stall timer of the release version of 800xA. Hence, the performance of our Java solution is acceptable.

[1] Tests conducted: 5.3.2:1-6, 10-14

4. **Testing the performance of the threads**
   Using our Java solution to create and start threads within the 800xA system was expected to take more time. To measure the extent of this, we first implemented the same thread functionality in C++ threads. To measure time, the following strategy was used: we used an 800xA internal timer, especially implemented for measuring short time spans and calculating average times. We decided to measure the time of a full cycle of the thread printing the MAC and IP addresses. We started the timer right after the printing, waited for the thread creation (at least) 3 seconds later and stopped the timer after the subsequent printouts. Between the printouts there are both method calls to our Java module and calls to the rest of the 800xA system through our bridge solution described in Section 5.1. Then we used the timer in the same way for the C++ threads started using *VOS*, as described in Section 5.2. The results can be seen in Figure 6.1. The Java solution introduces some additional method calls and was thus about 40 ms slower.

Table 6.1 Performance difference of our threads in Java and C++.

	Java	C++
Measurement 1	3.049 893 s	3.009 079 s
Measurement 2	3.048 052 s	3.007 834 s
Measurement 3	3.048 892 s	3.008 759 s
Measurement 4	3.048 875 s	3.008 262 s
Measurement 5	3.050 056 s	3.008 562 s
Measurement 6	3.049 497 s	3.008 546 s
Measurement 7	3.049 963 s	3.008 561 s
Measurement 8	3.048 708 s	3.008 554 s
Measurement 9	3.048 784 s	3.008 579 s
Measurement 10	3.050 828 s	3.008 711 s
Average time	3.049 355 s	3.008 544 s
Standard deviation	0.789 ms	0.308 ms

5. **Testing the memory footprint**
An interesting characteristic to evaluate is the memory footprint of our solution. We obtained this information by looking at the size of the heap of the system. The memory situation is shown in Figure 6. The first part of the memory is used by initial static allocations of the system, for instance the actual code and global variables. The LJRT solution with its static allocations of classes and its heap is allocated next. The size of the heap is determined by the configuration in `config.h`. After that, VxWorks is initialized and allocates a system pool for the operating system to use. Then the rest of the available memory is allocated as the heap of the system. Finally, the system pool grows, taking memory from the heap. By looking at the size of this system heap at a specific time during startup in our Java and C/C++ solutions, we could calculate how much memory overhead LJRT introduced. We found that this value was 52 kB for our Java solution.

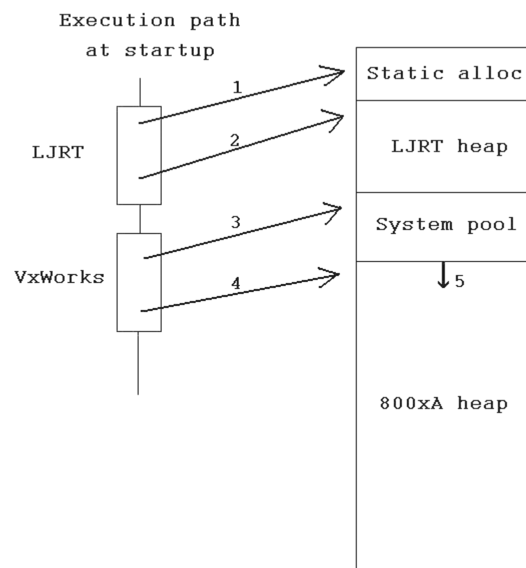

Figure 6.1 The memory situation after the startup sequence.

7. Conclusions

Developing large scale safety critical software is a very costly task. It is therefore desired to introduce languages with higher intrinsic safety in order to preclude common error types. The reduction of errors can save considerable amounts of verification costs and can ease certification processes. Safer languages also improve productivity, since less time is spent on the difficult work of detecting the origins of grave bugs, like the ones resulting from poor manual memory management.

This object of this thesis was to determine whether there were any suitable alternatives to the currently used implementation language of ABB 800xA. Today, C/C++ is used consequently throughout the system, languages without a safety focus. We have determined the most relevant factors to be judged when choosing a new implementation language for a system like ABB 800xA. We found that the following factors are the most interesting:

- Productivity
- Safety
- Concurrency
- Determinism
- Memory footprint
- Performance
- Portability

Furthermore, our thesis included a comparison between six different language solutions, in which we evaluated how well the factors were fulfilled. The languages we investigated were: C/C++, since they were used in the original implementation, Java, a language with an industrial acceptance, and Ada, specifically developed for safety related programming. We found that Java should be a good alternative to the current C/C++ solution, either by using a virtual machine implementing RTSJ or by compiling it to native code using LJRT. To utilize the virtual machines requires more memory and the LJRT solution has worse performance, what best suits the 800xA system is open to question. Using Ada is another option, and it gets little attention in this thesis due to too many biased sources combined with our initially chosen priorities. Ada is currently used in rather limited domains of the industry, but it has supporters. The future will show whether the language will have any major impact on the industry or not.

Our last part of this work dealt with introducing Java in the lower level parts of the ABB 800xA system, using LJRT. We have ported a part of the subsystem PM Redundancy called Role Selection and made it execute together with the original C/C++ solution. We tested the functionality using a formal test suite for regression testing with good results. We have thus showed that it is possible to introduce Java in the controller using LJRT, which is the main contribution of this thesis. As a final step we looked at the performance differences and the memory footprint of our solution. The measurements gave us that the Java solution was slower; about 40 ms slower thread loops were detected. We also obtained that the additionally required memory using LJRT, in the order of 50 kB, a size which did not negatively influence the rest of the system.

7.1 Future work

Much can still be done to simplify and improve future Java solutions for 800xA using the LJRT solution. We point out some specific improvements.

Future releases of LJRT should solve some of the problems we had to solve manually. For example, the generated code should only use C comments. Proper support for VxWorks threads should also be included.

Although LJRT supports incremental garbage collection, there is no implementation for VxWorks. Due to our time constraints, we did not implement the garbage collector thread. What is left is a garbage collector of "stop the world" type. This has two main consequences. Most importantly, the GC is not deterministic, which can cause severe problems in a real-time system. Secondly, our system without the synchronization needed for a proper real-time GC has better performance than a solution with a functional incremental GC would have had. If support for VxWorks threads is added in the future, the incremental GC will work and the performance measures we made will then have to be repeated.

Using Visual Studio for compiling the code generated from LJRT, we had to do many configurations of the development environment, for example adding include directories and adding compiler flags. Also, the entire development cycle was somewhat lengthy. First, the written Java code was compiled by LJRT using the tool Make. Then we had to make changes in the generated C code. As the third step the code was loaded in Visual Studio together with the the rest of the 800xA source code and possibly new changes were made. Then the code was compiled and linked from Visual Studio and, finally, the binary was downloaded to the hardware for testing. Figure 7.1 illustrates the work involved every time a change of the Java code was to be tested. In the future, a tool could be developed that tightens the development cycle. The tool could also at least partly generate files similar to our bridge solution using the Java code as input, if such a design would be needed.

If all this would be implemented in the future, using LJRT would be a very useful way to gradually introduce Java in an existing C/C++ solution of 800xA.

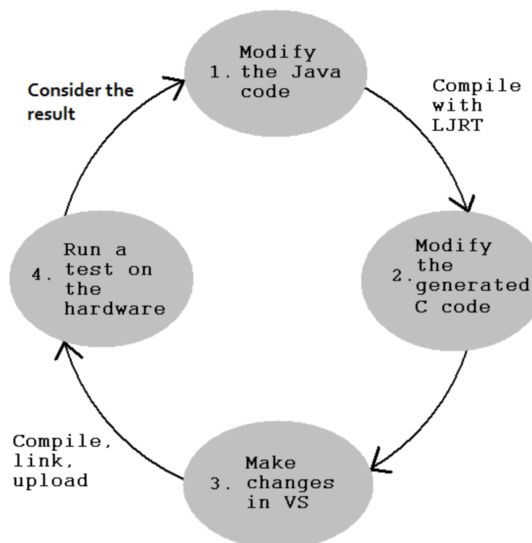

Figure 7.1 The development cycle.

8. Bibliography

[AG02] Tor Andræ and Håkan Gustavsson. Utökat javastöd i styrsystem. Master's thesis, Dept. of Computer Science, Lund Institute of Technology, Lund University Sweden, 2002.

[Aic07] Aicas. JamaicaVM 3.0 - User Documentation. Webpage, http://www.aicas.com/jamaica/doc/html (2007-04-03), 2007.

[Ang07] Ola Angelsmark. FTTD PM Redundancy. Function type test description, ABB/XA/ACB, May 2007.

[Arv07] Maja Arvehammar. Object Oriented Automation System. Master's thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund University Sweden, 2007.

[BBD+00] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-time Specification for Java*. Addison-Wesley, June 2000.

[Bro07] Ben Brosgol. Ada and Java: real-time advantages. Webpage, http://www.embedded.com/showArticle.jhtml?articleID=16100316 (2007-03-15), 2007.

[BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, Essex, third edition, 2001.

[BW06] Ben Brosgol and Andy Wellings. A Comparison of Ada and Real-Time Java for Safety-Critical Applications. In *Reliable Software Technologies - Ada Europe 2006*, June 2006.

[Con07] Marin David Condic. Is there any scientific evidence that indicates Ada is more productive and/or safer than other languages? Webpage, http://www.adapower.com/index.php?Command=Class&ClassID=FAQ&CID=327 (2007-03-08), 2007.

[Dij68] Edsger Dijkstra. Go To Statement Considered Harmful. Technical report, Communications of the ACM, Vol. 11, No. 3, March 1968.

[Dun07] John Duncan. What specific features does Ada have that makes it reliable? Webpage, http://www.adapower.com/index.php?Command=Class&ClassID=FAQ&CID=325 (2007-03-08), 2007.

[Eck02] Bruce Eckel. *Thinking in Java*. Pearson Education, Prentice Hall PTR, third edition, 2002.

[Eck07] Bruce Eckel. Correspondence via email, May 2007.

[Eve07a] EventHelix.com Inc. Comparing C++ and C (Inheritance and Virtual Functions). White Paper, http://www.eventhelix.com/RealtimeMantra/basics/ComparingCPPAndCPerformance2.htm (2007-04-16), 2007.

[Eve07b] EventHelix.com Inc. Keep It Simple. White Paper, http://www.eventhelix.com/RealtimeMantra/KeepItSimple.htm (2007-04-16), 2007.

[Fre07] Free Software Foundation. GNAT. Webpage, http://www.gnu.org/software/gnat/gnat.html (2007-08-10), 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Pearson Education, Prentice Hall PTR, third edition, 2005.

[Gre07] Green Hills Software. AdaMULTI. Webpage, http://www.ghs.com/products/AdaMULTI_IDE.html (2007-08-10), 2007.

[Hat95] Les Hatton. *Safer C: Developing Software for High-integrity and Safety-critical Systems*. McGraw-Hill Book Company Europe, England, first edition, 1995.

[Hel06] Markus Helgesson. DoF PM Redundancy. Description of function, ABB/XA/ACB, November 2006.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Dept. of Computer Science, Lund University, 1998.

[HS02] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual*. Pearson Education, Prentice Hall PTR, fifth edition, 2002.

[IEE07] IEEE POSIX Certification Authority. Portable Operating System Interface. Webpage, http://standards.ieee.org/regauth/posix/index.html (2007-08-19), 2007.

[Ive03] Anders Ive. *Towards an embedded real-time Java virtual machine*. Licentiate thesis, Dept. of Computer Science, Lund Institute of Technology, 2003.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Pearson Education, Prentice Hall PTR, second edition, 1999.

[Mic04] Sun Microsystems. The Real-Time Java Platform. Technical White Paper, 2004.

[NEN02] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. Real Java for Real Time – Gain and Pain. In *Proceedings of CASES-2002*, pages 304–311. ACM, ACM Press, October 2002.

[Nil06] Anders Nilsson. *Tailoring native compilation of Java for real-time systems*. PhD thesis, Dept. of Computer Science, Lund University, May 2006.

[NP97] Patrick Niemeyer and Joshua Peck. *Exploring Java*. O'Reilly, second edition, 1997.

[Pro07] Productive Robotics at Lund Institute of Technology. LJRT - Lund Java based Real-Time. Webpage, http://www.robot.lth.se/java/ (2007-08-19), 2007.

[RD06] José Ruiz and Robert Dewar. Ada for Embedded Programming. White Paper, Adacore, September 2006.

[Red07] Red Hat, Inc. Cygwin. Webpage, http://www.cygwin.com (2007-06-11), 2007.

[Rit93] Dennis M. Ritchie. The Development of the C Language. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, April 1993.

[Rob06] Sven Gestegård Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Dept. of Computer Science, Lund University, June 2006.

[Rog06] Patrick Rogers. Programming real-time with Ada 2005. *http://www.embedded.com/showArticle.jhtml?articleID=192503587*, September 2006.

[Årz03] Karl-Erik Årzén. *Real-Time Control Systems*. Dept. of Automatic Control, Lund Institute of Technology, Lund, 2003.

[Str93] Bjarne Stroustrup. A history of C++: 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, 1993.

[Ven98] Bill Venners. Designing with interfaces. Webpage, http://www.javaworld.com/jw-12-1998/jw-12-techniques.html (2007-08-19), 1998.

[Wei03] Marcus Weiskirchner. Comparison of the Execution Times of Ada, C and Java. Paper, September 2003.

[Whe07] David A. Wheeler. Ada, C, C++, and Java vs. The Steelman. Webpage, http://www.adahome.com/History/Steelman/steeltab.htm (2007-03-15), 2007.

[You82] S. Young. *Real-time languages: Design and Development*. Ellis Horwood, 1982.

A. Specific modifications to integrate Java

Our experimental setup for the development was a PC with an Intel CPU and Microsoft Windows XP SP2 and the following software:

- LJRT - Lund Java based Real-time, development version from February 2007
  `http://www.robot.lth.se/java/`
- Cygwin, CYGWIN_NT-5.1 1.5.24(0.156/4/2)
  `http://www.cygwin.com`
- Microsoft Development Environment 2003, version 7.1.3088
  `http://msdn.microsoft.com`
- Tornado 2.2.1/VxWorks 5.5.1
  `http://www.windriver.com/support/resources/`
  `tornado22_bulletin.html`

A.1 Details for Role Selection

To achieve our results the following strategy was used: We started by generating C code from our Java code using LJRT. The old source files corresponding to the Role Selection module of 800xA were removed from the project in Visual Studio. The next step was to add the generated LJRT output, the necessary files of the LJRT solution, and make some modifications. The complete list of files we used in the project to make Role Selection work are presented below:

- Source files for 800xA, except the ones for the module Role Selection.
- The LJRT generated C file and its corresponding header files.
- The files with native code.
- The files needed for the bridge we implemented, for function calls to and from our Java module (as described in Section 5.1).
- The following files from the LJRT library 'gc': `gc_incremental_ms.c`, `gc_roots.c`, `gci_common.c`, `gci_sched.c`, `gci_thread.c`
- The following files from the LJRT library 'javanative': `java_lang_string.c`, `javatypes.c`

A number of changes were required to make everything compatible enough for succesful compilation and linking in Visual Studio. Much of the work involved in this thesis dealt with unexpectedly sharp edges between the different components we used. Some of the issues would not have appeared if we had worked with a compiler compliant to the C99 standard. Specific changes we made in the source code of LJRT and 800xA included:

- Our compiler could not handle the C99 syntax of macros with a variable number of arguments, included in LJRT files. To make the code compile, we had to change all occurrences to conform with the syntax of the variadic macros of GCC. For example:

```
#define debug(format, ...)
fprintf (stderr, format, __VA_ARGS__)
```

was changed to:

```
#define debug(format, args...)
fprintf (stderr, format, args)
```

- To enable cross-compiling for VxWorks, we needed to a add a new target in the `configure.in.skel` file of LJRT. The corresponding argument used when configuring LJRT was then `-host=intel-i386-vxworks`
- Support for one-line comments beginning with '`//`' was an addition to C99. Consequently, in all LJRT C-files included in the project in Visual Studio, we had to change all those comments to standard C comments (`/* ... */`).
- We removed the occurrences of the macro `MONITOR_INIT` used for instance during initialization of classes in the generated C file. This was done by redefining the macro to nothing:

```
#define MONITOR_INIT(obj)
```

- Our different environment and thread solution meant we could remove the various inclusions of `pthread.h` and `sys/time.h` throughout the files included in the project.
- To properly initialize the Java classes, the garbage collection, and the threading behaviour we needed to include the code below at an appropriate position during the start-up of 800xA. In our final solution we put it immediately after the initialization of the global variables of the entire system.

```
GC_INIT(GC_HEAPSIZE);
{
   thread_count = 0;
   javaClasses_Init(__eFlag__);
}}}
```

Two changes have to be made after every compilation of Java code:

- The default configuration allocates two much memory for the Java heap. GC_HEAPSIZE in `config.h` had to be lowered. We settled for a heap size of 100000 B. This change is however only needed if a LJRT reconfiguration was done before the compilation.
- Remove C++-comments from the generated C-file, appearing at three positions. Again, this would not be necessary if a C99 compiler was used.

Three changes in the configuration of the project in Visual Studio was needed as well.

- Four new additional include directories were needed. The *directory with the Java source files (and native sources)*, the *build directory* where the LJRT output was generated, and the *gc* and *javalib/native* directory of LJRT.
- To enable the configuration provided by the file `config.h`, the preprocessor definition `HAVE_CONFIG_H` was added.
- Internal ABB tools increase the severity of warnings and force the compiler to treat them as errors. We disabled the warnings for some of our source files using the flag '`-w`'.

A.2 Details for Java threads

In this section, the work involved with the making the Java threads execute in our environment is described. It is assumed that the experimental setup is as described in Appendix A and that the modifications in Appendix A.1 already have been made. Another file had to be included in the LJRT library 'javanative': `ljrt_threading.c`. This file and the already included `gci_thread.c` (part of the LJRT library 'gc') needed many changes to circumvent the Pthread calls and use VOS instead as mentioned in Section 5.2.

We made the following major changes in the two files:

- `ljrt_threading.c`
  The existing version used Pthread semaphores for mutual exclusion during thread handling. We solved this part with the sema-phores of VOS instead.
- `ljrt_threading.c`
  We have modified the procedure `java_lang_Thread_start` so that `pthread_create` no longer is used for the thread creation. Instead the VOS procedure `vosThreadSpawn` is called for this purpose. A limitation with our solution is that we always create Java threads with the VOS priority level 'normal' and the functionality of `java_lang_Thread_inheritPriority` is not implemented.
- `ljrt_threading.c`
  Minor modifications in most functions and also in the structs `ljrt_thread_t` and `rundata_t`. In the end the following functions were present: `inc_threads`, `dec_threads`, `wait_for_all_threads_to_die`, `java_lang_Thread_createThread`, `java_lang_Thread_inheritPriority`, `ljrt_thread_start`, `java_lang_Thread_start`, `thread_init`, `runner` and `thread_start`. Functions not mentioned here were removed.
- `gci_threads.c`
  This file includes implementation details of the incremental garbage collector. The parts implemented for Pthreads were changed to a solution very similar to the one used for threads in STORK[1], available in the same file. This resulted in a working "stop the world"-GC when we used the VOS function `vosThreadGetMyId` to get IDs of specific threads and `vosThreadDelete` for deletion.

[1]A real-time kernel developed by the Department of Automatic Control, Lund Institute of Technology.

B. LJRT configuration file

Below is the appearance of the file `config.h`, generated in the LJRT build directory, after our modifications.

```
/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.in by autoheader.  */
/* Set to turn on free block coalescing (only for nonmoving GC) */
/* \#undef ALLOC_COALESCE */
/* Define when building with BITS64 */
#define BITS64 1
/* Set to enable cpu logging */
/* #undef CPU_UTILIZATION_LOGGING */
/* Which GC algorithm to use. see gc.h for details */
#define GC_ALGORITHM 2
/* set to use auto-tuning GC */
/* #undef GC_CYCLE_TIME_ADAPTIVE */
/* Set to turn on coalescing of contihous dead objects b/f free
(only for nonmoving GC) */
/* #undef GC_FREE_CONTIGOUS_HACK */
/* Set to turn on GC finalizers */
#define GC_HAS_FINALIZERS 1
/* Set if threads enabled */
#define GC_HAS_THREADS 1
/* Size of GC heap */
#define GC_HEAPSIZE (100000)
/* Set to run GC incrementally */
#define GC_IS_INCREMENTAL 1
/* Set to enable GET_THREAD_ROOT optimization */
#define GC_LAZY_GET_THREAD_ROOT 1
/* EXPERIMENTAL */
#undef GC_MARK_ROOTS_ATOMIC
/* set to measure S_h */
/* #undef GC_MEASURE_HEAPSTATE */
/* Set to turn off the debug layer */
#define GC_NODEBUG 1
/* EXPERIMENTAL */
/* #undef GC_NO_GC_LOCKING */
/* set to use only time and no work metric */
/* #undef GC_ONLY_TIME */
/* set to predict C_GC after mark phase */
/* #undef GC_PREDICT_C */
/* Set to turn on GC warnings */
/* #undef GC_PRINT_DEBUG */
/* Set to turn on GC warnings */
/* #undef GC_PRINT_WARNINGS */
/* Set to turn on GC profiling */
/* #undef GC_PROFILING */
/* Thread model. see gc.h for details */
#define GC_THREAD 1
/* Set to reference call ref params */
#define GC__REF_CALL_PARAMS 1
/* Set to turn on exceptions. See exception.h for details */
#define HAS_EXCEPTIONS 2
/* Define when building with HAS_FLOAT */
```

```
#define HAS_FLOAT 1
/* Set to turn on reflection */
#define HAS_REFLECTION 1
/* Define when building for AVR */
/* #undef HOST_AVR */
/* Do memory initialization in allocator on allocation
(only for nonmoving GC)*/
/* #undef INIT_ON_ALLOC */
/* Do memory initialization in allocator on free
(only for nonmoving GC) */
/* #undef INIT_ON_FREE */
/* Do memory initialization in GC (only for nonmoving GC) */
/* #undef INIT_ON_GC */
/* set to enable Java-specific features */
#define JAVA_2_C 1
/* Define when building with LINUX */
/* #undef LINUX */
/* set to enable logging (only works with ivm threads) */
/* #undef LOGGING */
/* set to enable non-critical memory */
/* #undef LPMEM */
/* set to measure C_GC */
/* #undef MEASURE_CGC */
/* Name of package */
#define PACKAGE "RoleSelection"
/* Define to the address where bug reports for this package
should be sent. */
#define PACKAGE_BUGREPORT ""
/* Define to the full name of this package. */
#define PACKAGE_NAME ""
/* Define to the full name and version of this package. */
#define PACKAGE_STRING ""
/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME ""
/* Define to the version of this package. */
#define PACKAGE_VERSION ""
/* Set to use powers-of-two block sizes
(only for nonmoving GC) */
/* #undef POWER_OF_TWO_ALIGN */
/* Define when building with RTAI */
/* #undef RTAI */
/* Set to turn on heap and pointer sanity checks
(only for nonmoving GC) */
/* #undef SANITY_CHECKS */
/* Define to 1 if you have the ANSI C header files. */
#define STDC_HEADERS 1
/* set to enable GC time measurements */
/* #undef TIME_MEASUREMENTS */
/* set to use the time-triggered GC */
/* #undef TIME_METRIC */
/* Version number of package */
#define VERSION "autotools-0.0.1"
/* Define when building with LINUX */
#define VXWORKS 1
/* Set to enable AVR external clock */
/* #undef _AVR_EXT_CLOCK */
```

```
/* Set to enable AVR external memory */
#define _AVR_EXT_MEM 1
/* Set to enable UART in the AVR */
/* #undef _AVR_UART */
/* Normal Java thread priority */
#define _RT_NORMAL_PR_JAVA (5)
/* RT Scheduling policy */
#define _RT_SCHEDULING_POSIX (2)
/* Define to empty if ‘const’ does not conform to ANSI C. */
/* #undef const */
```