Final Thesis

Run-time environment for Plex-C on JVM

by

Johan Möller

LITH-IDA-EX--02/100--SE

2002-10-31
Final Thesis

Run-time environment for Plex-C on JVM

by

Johan Möller

LITH-IDA-EX--02/100--SE

2002-10-31

Supervisors: Karl-Johan Karlsson and Leif Larsson
Examiner: Christoph Kessler
Abstract

The Ericsson AXE-based systems are programmed using an internally developed language called Plex-C, which is specifically designed for that purpose. Plex-C is normally compiled to execute on an Ericsson internal processor architecture, which uses a specialized set of instructions. A transition to standard processors is currently in progress. This makes it interesting to examine if Plex-C can be compiled to execute on the JVM, which would make it processor independent.

The purpose of the thesis is to examine if parts of the run-time environment of Plex-C can be translated to Java and if this can be done so that sufficient performance is obtained. It includes how language constructions in Plex-C can be translated to Java.

The thesis describes how a limited part of the Plex-C run-time environment is implemented in Java. Since the AXE is a real-time system, the performance is very important. Java optimizations are therefore an important part of the implementation.

It is also described how the JVM system was tested with a benchmark test. This test is specifically developed by Ericsson for measuring performance of emulations of their AXE platform.

The test results indicate that the implemented system is a few times faster than the Ericsson internal processor architecture. But this performance is still only approximately 50% of the speed of the tested version of a new Ericsson system that is under development for Tru64 running on Compaq Alpha processors.

To be an interesting replacement for the currently used processor architecture, the speed of the JVM system would have to be approximately at least 70% of the finished version of the new tested Ericsson system. This means that the JVM is most probably not suitable for further analysis as a replacement for the currently used processor architecture. It might still be useful as a processor independent test platform.
## Contents

1 Introduction 1  
1.1 Background 1  
1.2 Assignment 1  
1.3 Purpose 3  
1.4 Structure 3  
1.5 Requirements 4  

2 AXE 7  
2.1 Overview 7  
2.2 APZ 8  
2.3 Central Processor 8  
2.3.1 Register Memory 9  
2.4 Main memory 9  
2.4.1 Reference Store 10  
2.4.2 Data Store 10  
2.4.3 Program Store 10  
2.4.4 Job Buffer Memory 10  
2.5 Regional Processor 10  
2.6 APZ VM 11  

3 Plex 13  
3.1 Blocks 13  
3.2 Variables 14  
3.3 Variable types 15  
3.3.1 Field variables 15  
3.3.2 Symbol variables 15  
3.3.3 String variables 16  
3.4 Variable properties 16  
3.4.1 Storage properties 17  
3.4.2 Start/restart/reload properties 17  
3.4.3 Function change properties 19  
3.5 Variable addressing 19  
3.6 File 19  
3.6.1 Record 20  
3.6.2 Pointer 20  
3.6.3 File size alteration 20  
3.7 Signals 21  
3.8 Signal data 21  
3.9 Signal properties 21  
3.9.1 Unique / Multiple 22  
3.9.2 Direct / Buffered 22  
3.9.3 Single / Combined 23  
3.10 Signal types 24  
3.10.1 CP-CP signals 24  
3.10.2 RP signals 24  
3.11 Signal addressing 25  
3.12 Job handling 26
3.12.1 Job buffers 32
3.12.2 Job table 32
3.12.3 Time queues 33

4 Solution 35
4.1 Blocks 35
4.2 Addressing 40
4.3 Variable types 41
  4.3.1 Java variable types 41
  4.3.2 Field variables 42
  4.3.3 Symbol variables 43
  4.3.4 String variables 44
4.4 Variable properties 44
  4.4.1 TEMPORARY 45
  4.4.2 DS 46
  4.4.3 BUFFER 46
  4.4.4 CLEAR 47
  4.4.5 RELOAD 47
  4.4.6 DUMP 47
  4.4.7 TRANSIENT 48
  4.4.8 STATIC 48
4.5 Files 49
4.6 Signals 49
4.7 Job handling 52
  4.7.1 Job buffers 52
  4.7.2 Job table 52
  4.7.3 Time queues 56
4.8 Signal properties 58
  4.8.1 Buffered signals 58
  4.8.2 Direct signals 59
4.9 Threads 59
  4.9.1 External interrupt 60
  4.9.2 Job execution with three threads 60
  4.9.3 Job execution with one thread 62
4.10 Subroutines 67
4.11 Synchronization 68
  4.11.1 Placement of interrupt indicators 69
  4.11.2 Interrupt flag or counter 70
  4.11.3 Placement of CIS counter 71
  4.11.4 Series of job table scans 72
  4.11.5 Job table flag 72
  4.11.6 Removal of CIS thread 72
  4.11.7 Optimization overview 73
  4.11.8 Deadlock analysis 73
4.12 Other design issues 76
  4.12.1 Date conversion 76
  4.12.2 Robust 77
4.13 Usage 77
  4.13.1 Regional processors 77
  4.13.2 Central processor 77
4.14 Class descriptions 78
Figures

FIGURE 1. Architectures with APZ and APZ VM compared to thesis work 2
FIGURE 2. Run-time environment compared to code library when interacting with application 5
FIGURE 3. Overview of some central parts of an AXE exchange 7
FIGURE 4. Organization of memory stores 9
FIGURE 5. Variable addressing 19
FIGURE 6. Difference of unique and multiple signals 22
FIGURE 7. Difference of direct and buffered signals 23
FIGURE 8. Difference of single and combined signals 23
FIGURE 9. The different types of software signals 24
FIGURE 10. Signal sending 25
FIGURE 11. The different priority levels of jobs 27
FIGURE 12. Overview of job handling 29
FIGURE 13. Example of job handling 31
FIGURE 14. Sending a delayed signal 33
FIGURE 15. Overview time queues 34
FIGURE 16. Queue with signals and their destinations 37
FIGURE 17. References using a table 37
FIGURE 18. Block design in Java 40
FIGURE 19. Signal design in Java 51
FIGURE 20. Different table approaches 56
FIGURE 21. Execution of a C- or D-level job (when interrupts are on) 63
FIGURE 22. Main-loop (executed by Main thread). Text in italics must be synchronized 65
FIGURE 23. Yield-method for C level jobs (executed by Main thread). Text in italics must be synchronized 66
FIGURE 24. Yield-method for D-level jobs (executed by Main-thread). Text in italics must be synchronized 67
FIGURE 25. Loops after some optimizations (compare to Figure 22, 23, and 24) 73
FIGURE 26. Locks by CIS thread when CIS interrupt occurs 74
FIGURE 27. Locks by RP thread when queuing external signal 75
FIGURE 28. Locks by the job thread to synchronize interrupt flags during execution of the main loop 75
FIGURE 29. Locks by the job thread during fetch of a signal from the job queue 75
FIGURE 30. Locks by job thread when queuing an internal signal 75
FIGURE 31. Locks by job thread during scanning of job table 76
FIGURE 32. Class diagram 79
FIGURE 33. EB test blocks 85
FIGURE 34. A successful call 88
FIGURE 35. A blocked call 89
FIGURE 36. A busy or not found call 89
FIGURE 37. Example of a call in progress 89
FIGURE 38. Test case EB1 measured with fast (diamond) and classic (triangle) JVM 95
FIGURE 39. Test case EB1 measured using fast32 (diamond) and fast64 (triangle) 96
FIGURE 40. Test case EB1 measured with copying (diamond) and compacting (triangle) GC 98
FIGURE 41. Garbage collection times for copying (diamond) and compacting (triangle) GC. Test case EB1 98
FIGURE 42. Test case 1 measured with JVM (diamond) and APZ VM (triangle) 102
FIGURE 43. Test case 2 measured with JVM (diamond) and APZ VM (triangle) 102
FIGURE 44. Test case 3 measured with JVM (diamond) and APZ VM (triangle) 103
FIGURE 45. Test case 4 measured with JVM (diamond) and APZ VM (triangle) 103
FIGURE 46. Test case 5 measured with JVM (diamond) and APZ VM (triangle) 104
FIGURE 47. Test case 6 measured with JVM (diamond) and APZ VM (triangle) 104
FIGURE 48. Test case 7 measured with JVM (diamond) and APZ VM (triangle) 105
FIGURE 49. Test case 8 measured with JVM (diamond) and APZ VM (triangle) 105
Tables

TABLE 1. Possible combinations of variable properties 16
TABLE 2. Relevant Java primitive types 41
TABLE 3. Implementation of Plex field variables in Java 43
TABLE 4. Parameters for the test 91
TABLE 5. Test cases 92
TABLE 6. Garbage collection times for copying and compacting GC. Test case EB1 99
TABLE 7. Speed of the JVM compared to the APZ VM 106
TABLE 8. Test case EB1 measured with fast and classic JVM 141
TABLE 9. Test case EB1 measured using fast32 and fast64 141
TABLE 10. Test case EB1 measured with copying and compacting GC 141
TABLE 11. Test cases measured under JVM and APZ VM 142
Run-time environment for Plex-C on JVM
1 Introduction

1.1 Background
The Ericsson AXE platform is used for things like exchanges in telephone networks and central traffic-systems in cellular networks. To program the AXE-based systems, an internally developed language called Plex-C is used. Plex-C is normally compiled to execute on an Ericsson internal processor architecture, which uses a specialized set of instructions. But Ericsson is currently transitioning to standard processors.

Plex-C is specifically designed for the AXE platform, which makes the language very specialized. The Plex-C applications need a run-time environment to execute.

The thesis work was performed at Ericsson SoftLab AB.

1.2 Assignment
The development for standard processors makes the JVM (Java Virtual Machine) interesting. If the system could run on the JVM it would be processor independent. Many companies are putting lots of effort in the development of their freely available JVMs, which could be used.

Since the use of the JVM would make the system platform compatible, all applications can be moved to another platform that has a good JVM implementation without any need to recompile or rewrite the source code. The underlying processor architecture could then easily be exchanged if necessary.

The primary goal of the thesis work was to examine if parts of the Plex-C run-time environment could be translated to Java and if that run-time environment could have sufficient performance.

The thesis work includes both design of the run-time environment and translation of some Plex-C language constructions.

The run-time environment itself is not useful. It must be complemented with a Plex-C to Bytecode compiler. It is therefore important that the run-time environment provides a simple and efficient interface that can be used by compiled applications.
The compiler is not included in this thesis work, but might be implemented in a future thesis work.

A new Ericsson system that is under development, called APZ VM, runs on Tru64 that uses Compaq Alpha processors. The performance of the thesis work will be compared to this system. The thesis work will be tested on the same OS and processors as the APZ VM, which is a fast multiprocessor system. Compaq has however decided to discontinue with the Alpha processors.

APZ VM is a virtual machine that emulates the old APZ processor. The thesis work primarily focuses on how the APZ works, but since the APZ VM is able to substitute the APZ it may also be considered if needed.

The run-time environment that is implemented in the thesis work will run on the JVM. The APZ VM contains the run-time environment, which gives that run-time environment advantage over the JVM run-time environment by having direct access to the OS. An overview of the three systems is shown in Figure 1. Applications for the APZ and APZ VM are compiled to ASA code and applications for the JVM are compiled to Bytecode.

![Figure 1. Architectures with APZ and APZ VM compared to thesis work](image)

The APZ VM is specifically written and optimized for the Tru64 OS running on Alpha processors, with a run-time environment with direct access to the OS. The run-time environment implemented in the thesis will therefore most probably be slower. The JVM also has the speed disadvantage of being designed for a general OS, as opposed to the APZ VM which is designed for Tru64. The goal is not to make an equally fast system, it is to see how fast a portable environment can be compared to the real system and if this speed is sufficient to make it useful.
1.3 Purpose
The purpose of this report is to describe the thesis work and present the achieved results.

It will hopefully also be a useful resource in completion of the run-time system and for creating a Plex-C to Bytecode compiler.

1.4 Structure
The thesis consists of the following 6 chapters.

Chapter 1 (Introduction) provides an introduction to the thesis.

Chapter 2 (AXE) presents the Ericsson AXE platform.

Chapter 3 (Plex) presents the Ericsson Plex-C programming language.

Chapter 4 (Solution) describes how the thesis assignment was solved.

Chapter 5 (Test) describes how the solution was tested and the results of the tests.

Chapter 6 (Results) gives the results of the thesis work.

Persons with good knowledge of AXE and Plex-C can skim through Chapters 2 and 3. They contain information that is necessary for understanding Chapters 4 and 5.

Chapter 4 and 5 are sometimes very detailed and can be read to the extent of interest.

If only the results are interesting, it suffices to read Chapter 6.

The reader is assumed to have a general knowledge of the Java language, the JVM, and the Java API. Specialized concepts within Java are however always explained before they are used. A few comparisons to the C programming language are also made.
1.5 Requirements

The goal of this thesis is mainly to see if it is possible to use JVM as a platform for Plex-C. To actually see if this is possible, the run-time environment needs to implement most of the functionality mentioned in Chapter 2 and 3.

The system is however far too big to be fully considered in this thesis. The thesis is therefore focused on examining the possibility of using the JVM and implementing the most necessary functions for measuring of the performance. Basically this was to consider some parts of the Plex-C concepts of blocks, signals, job queues, and Data Store (DS). Parts that should not be considered were mainly the Plex-C concepts of Central Processor (CP) redundancy (see Section 2.3) and function change (see Section 3.4.3).

To make a fair judgement of the performance capabilities of the JVM system, it is very important to use speed optimizations in the implementation. Part of the thesis is therefore focused on this. Specific optimizations should however only be done as long as major design is not destroyed, otherwise the system will be hard to extend and maintain.

Java is often harder to optimize compared to other programming languages. This is mostly because of threads and exception semantics. The requirement of platform independency also makes optimization harder, since optimizing code for one platform may actually make it worse on another platform.

Although the real issue is to compile Plex-C to Bytecode, Bytecode will not have to be considered in the thesis work. Since the Java language is so closely related to the Bytecode it is sufficient to translate to Java and then compile to Bytecode with a Java to Bytecode compiler instead. Bytecode is however in general more powerful than Java.

The run-time environment should provide an interface to the compiler, with above mentioned limitations. Parts of the thesis therefore consist of deciding which parts that will be included in the run-time environment and which parts that will be generated by the compiler.

A run-time environment acts in some ways similar to a code library. It is similar in that it provides methods that can be called from the application. But the run-time environment is started before the application, which in turn is started by the run-time
environment. The run-time environment controls the execution of the application, since it is executed inside the run-time environment. Figure 2 shows the differences when an application is *started*, making *method call*, and at last *finished*. This is also a matter of implementation, the startup code could be made as a subroutine call of the application.

**Figure 2.** Run-time environment compared to code library when interacting with application.
Introduction
2 AXE

The AXE exchange is designed by Ericsson. It is used as a platform for several different nodes in telephone networks. The currently used AXE exchange is AXE 10. The descriptions in the thesis are targeted to AXE 10.

This chapter describes those parts of an AXE exchange that are relevant for the thesis work.

2.1 Overview

An AXE exchange is controlled by two concurrently executing CPs (Central Processors). RPs (Regional Processors) are connected to the CPs through a bus. The RPs do not communicate directly with each other. There are also another kind of RPs, called EMRPs (Extension Module Regional Processors), which are connected to the CPs through a pair of buses and a bus converter. The RPs and EMRPs control the hardware of the application. The hardware consists mainly of switches. This simplified overview of some central parts of an AXE exchange is modelled in Figure 3.

Programs for the CPs are written in a Plex dialect called Plex-C. This code is compiled into ASA 210C assembler. Sometimes also parts of the programs for the CPs are written directly in ASA 210C assembler.
RP programs are often written directly in assembler code. The used assembler code is ASA 210R or ASA 21R.

Programs for the EMRPs are often written in a different Plex dialect called Plex-M, which is an 8-bit version of Plex. This code is compiled into Motorola ASM6809 assembler.

This thesis only treats programs written for the CPs. Only programs written fully in Plex-C will be considered.

2.2 APZ
The classic APZ is a special purpose hardware developed by Ericsson. It is currently used in the AXE exchanges.

The major parts of the APZ are the CPs and the main memory.

2.3 Central Processor
For error recovery reasons, there are two CPs in the exchange. They both execute all central software in parallel and at regular intervals compare their results. Only one of the processors controls the exchange at a time. If the comparison fails an error recovery procedure is started, which may lead to switching of the controlling processor. The eventual switching is made by a special maintenance unit.

Since only one processor has the central control, they can from outside be treated as only one processor. This abstraction is followed in most of the remaining parts of this thesis.

The CP handles access to the main memory and also has internal registers [19]. The internal registers are described in the following section.

The CP also handles sending and receiving of signals. If the signals need to be temporarily stored this is also handled by the CP. The signal concept will be more explained in Chapter 3.

Communication with the regional processors is also handled by the CP.
2.3.1 Register Memory
There are three different priority group levels where programs can execute.

Each of the priority group levels has an own set of registers in the RM (Register Memory). Most of these registers are work registers used in calculations and for storing temporary variables. The reason for having register sets for each group level is to make it fast to do context switching between priority group levels. This means that no data has to be copied to the stack when switching, since the other priority group level does not use the same register set. Context switch between programs at the same priority group level or to a lower priority group level is not allowed.

There of course also exist other special registers which are not mentioned here. The interested reader can study reference [1].

2.4 Main memory
The CP dedicates some part of the main memory to store specific memory areas.

The main memory has three memory areas which are called stores. They are used for storing data and programs. The stores consist of the RS (Reference Store), the DS (Data Store), and the PS (Program Store).

The main memory also has a memory area called JBUM (Job Buffer Memory) ([19]). This area is used for storing buffered signals (see Section 3.9.2).

The different memory areas are further explained in the following sections. A simple overview is provided in Figure 4.

All access to data and programs goes through the RS. There are several reasons for having memory organized in this way.
• Programs can be replaced by other programs just by changing the corresponding references in the RS. This is used for program corrections and function changes (see Chapter 3).

• The memory is protected. Programs can only access and modify their own memory.

2.4.1 Reference Store
The RS is a memory area used for storing references to the DS and the PS. All data and programs must be referenced from somewhere in RS. The RS contains information about where to find the different data and programs.

To organize the different stores around RS give possibilities for function change and program corrections.

To find data or programs the associated block number is needed. This information is used to read in the RS where data are located in DS or programs are located in PS.

2.4.2 Data Store
All data are stored in the DS. The data are arranged as variables. Each DS variable belongs to one block in the PS. The variables can only be accessed from this specific block.

2.4.3 Program Store
All programs are stored in the PS. The programs are grouped into blocks. Signal information for the blocks is also stored in the PS.

2.4.4 Job Buffer Memory
Buffered signals are stored in the JBUM. The JBUM is divided into different areas for signals with different priorities and types.

2.5 Regional Processor
The CP delegates simple and time-consuming tasks to the RPs. This is made to decrease load on the CP.

The RPs control different kinds of hardware devices using RP software.
2.6 APZ VM

The APZ VM is a software emulation of the APZ. It is designed to be a successor to the classic APZ. APZ VM is under development and is planned to be used in the AXE exchanges in the future.

Since the APZ VM emulates the APZ, it is backwards compatible with old ASA code. APZ VM does not always truly emulate the APZ. It does not support two processors that execute the same code in parallel and compare the results. Signal interrupts are not treated exactly as in the APZ.
3 Plex

In the 1970s Ericsson needed a programming language for use with their AXE exchanges. They did not find any language that met their requirements. This led to the development of an own language called PLEX (Programming Language for EXchanges). The language was later extended in 1983. PLEX is often written with only the first letter capitalized (Plex).

Plex is a high-level and real-time programming language. It is specifically designed for use with telephone systems. Telephone systems do not need all kinds of elements that exist in the usual programming languages. For this reason, concepts like while-loops, negative integers and real-numbers do not exist in Plex. But Plex instead makes need of specialized concepts such as signals. This makes it necessary for Plex to have a run-time environment.

There exist several different Plex dialects. Plex-C is the Plex dialect that is used in the CP. This is also the only Plex dialect that is treated in this thesis. If not otherwise specified, from now on this thesis always means the Plex-C dialect when only Plex is written.

Plex is very specific to the APZ processor ([19]) and even includes support for inlining ASA code directly.

This chapter only describes those aspects of Plex that are relevant for the thesis. For further information about the language, see references [1], [2], and [3].

The first section explains the concept of blocks, which are the building blocks of Plex. After that follow sections explaining the variables and signals that are used in Plex. Finally the chapter is ended with an explanation of the order in which Plex executes applications. This is realized by a job handling system.

3.1 Blocks

The code in the CP is grouped into modules called blocks. A block has data encapsulation and is separately loadable ([3]). Data encapsulation means that the variables of a block are private and can therefore not be accessed by other blocks. An application is a group of blocks that interact with each other and the system. The interaction between the blocks is done by sending signals.
A block is described in a source code document. Signal descriptions can be included in that document or they can be described in separate signal description documents. Each block is complemented with a signal survey document that describes which signals the block is able to receive and send. Further information about the signal description and the signal survey documents can be found in reference [4] and [5] respectively.

The source code for a block consists of these four parts (see also [1]).

- The declare sector contains the variable and constant declarations.
- The program sector contains the executable statements. They are divided into subprograms. A subprogram starts with a signal-receiving statement and ends with either an exit statement or a direct signal-sending statement. The execution of a subprogram starts when a signal is sent to it. There is no reason to place statements outside of the subprograms. They will never be executed anyway.
- The data sector contains initial values for variables and file sizes. The initialization is done when the block is loaded into the exchange.
- The ID sector contains identification for the document. This part consists only of comments, and is not compiled.

### 3.2 Variables

Variables are declared in the declare sector and initialized in the data sector.

As any ordinary programming language, Plex contains a number of different variable types. The variables can further have different properties specifying things like for example storage.

Plex has data structures called files. This name can sometimes be confusing since files usually are associated with files in a file system. But file in Plex means a data structure of an array containing structured variables.

The data initialization, which is specified in the data sector, is done when blocks are loaded into the exchange. If no value for a variable is set in the data sector the value will be unspecified, unless the variable has the CLEAR property.

Plex supports data encapsulation. Variables always belong to one block, and can almost never be directly accessed from other blocks. One exception to this is
variables declared with the BUFFER property, they can be accessed from other blocks once the other blocks get the variable references.

The scope of a variable is the whole block in which it is placed. This includes any subroutine or subprogram.

### 3.3 Variable types

There are three different variable types in Plex: Field variables for integers, symbol variables for enumerations, and string variables for strings. Variable properties are described in Section 3.4.

#### 3.3.1 Field variables

Field variables store unsigned integers. The numeric value of a field variable can be from 0 up to \(2^n-1\), where \(n\) is the length of the variable in number of bits. The length can be 1, 2, 4, 8, or 16 bits. Variables with the TEMPORARY property (explained in Section 3.4) always have a length of exactly 16 bits unless they are R variables. Special cases of field variables are R variables and arrays:

- **R variables.** The length of R variables is specified by the processor that loads the block. This makes it possible to use variables (including those with the TEMPORARY property) with a length greater than 16 bits. However, an R variable can not be used as an index for an array. All CPs can handle variables of 16 bits length, R variables increase this length on some processors.

- **Arrays.** Arrays consist of a number of variables that have the same variable types. The variable type can be any field variable type except an array. Arrays can not have the BUFFER or TEMPORARY property. When declaring an array both the variable type and the number of elements must be specified. The maximum number of elements is depending on the target machine. Arrays can be one- or two-dimensional.

#### 3.3.2 Symbol variables

Symbol variables store symbolic values, for example: idle, busy, blocked, false, and true. Symbol variables are known as enumeration type variables in the C programming language. A list of the possible values is specified at declaration. Symbol values are replaced with numbers during the code generation, corresponding to the order of the symbol values in the symbol value list (starting from 0 and upwards). Symbol variables always have the length 16 bits. This means that the maximum number of different symbolic values for one variable is 65536.
3.3.3 String variables
String variables store text strings as sequences of alphanumeric characters. String variables always have the DS property. The maximum number of characters the string variable should be able to contain is specified at declaration. The maximum number of characters a string can be declared to contain is 255. Characters are encoded using the ISO CCITT alphabet number 5, which uses seven bits for each character.

3.4 Variable properties
Variables in Plex can have a number of different properties. The possible properties are TEMPORARY, DS, BUFFER, CLEAR, RELOAD, DUMP, TRANSIENT, and STATIC. The properties can be divided into the groups shown below. The properties are described in detail in the following sections.

- **Storage** - TEMPORARY, DS, and BUFFER.
- **Start/restart/reload** - CLEAR, RELOAD, DUMP, and TRANSIENT.
- **Function change** - STATIC.

Not all combinations of the variable properties are possible. Table 1 shows all possible combinations (as defined in [2] and [3]).

Some documents do not consider TEMPORARY as a separate property since it is equivalent to the combination of not DS and not BUFFER.

### Table 1. Possible combinations of variable properties

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Field variable</th>
<th>Symbol variable</th>
<th>String variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS DUMP</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DS STATIC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DS RELOAD</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DS RELOAD DUMP</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DS RELOAD STATIC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DS CLEAR</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DS CLEAR DUMP</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>BUFFER</td>
<td>X</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>BUFFER DUMP</td>
<td>X</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>TEMPORARY</td>
<td>X</td>
<td>X</td>
<td>no</td>
</tr>
</tbody>
</table>
### 3.4.1 Storage properties

The following properties specify how the variable is stored.

- **TEMPORARY.** Variables with the TEMPORARY property are stored in the RM. They can only have values when their corresponding block is executing. Before they receive a value, they are undefined. The values are lost when the execution leaves the block. The big advantage with temporary variables is that they are fast to access because they are stored in the RM. They do not have to be fetched and written to the DS. There exists one set of registers for each priority group level. If execution of a block is interrupted, the temporary values are still saved. This is because the code which is interrupting is using another set of registers.

- **DS.** Variables with the DS property are permanently stored in DS. They need to be temporarily loaded into the RM when used, which makes them slower than temporary variables. The value is not lost when the execution leaves the block.

- **BUFFER.** Field variables can have the BUFFER property. They are dynamically allocated in DS. A buffer variable does almost not use any space in the DS when it is not allocated, the only space used is for the address of the buffer. After it is allocated it can be deallocated. Buffer variables are very slow, but allow access from other blocks. Other blocks gain access to the buffer variable when its reference is sent as a buffer pointer in a signal. A BUFFER variable consists of an array of integers, with 16 bits as the largest length (R variables are not permitted). During allocation, the number of elements is specified. BUFFER variables are deallocated during restart. This kind of variable is typically used for transferring large amounts of data between blocks.

### 3.4.2 Start/restart/reload properties

This is a very short description of the start, restart, and reload processes. The parts that not are relevant for the thesis are left out.

During the **start** of the system, the blocks are initially loaded into the AXE. This is followed by a **restart**. Then the exchange is ready to execute applications.
Before a restart the contents of all DUMP marked variables are saved into a dump file. During a restart all variables with the CLEAR property are reset. The content of the dump file is then printed to the screen. The printout is done for testing and tracing purposes.

At regular intervals, all variables with the RELOAD property are backed up into a file called system dump.

All commands that change the value of RELOAD variables are saved in files called command logs. These log files can later be used to trace and repeat the commands. A new log file is created after each system dump.

If a serious error occurs, the exchange can use system dumps and command logs to reload the variables with RELOAD properties to set the exchange into an older state. This is followed by a restart. After that the exchange is again ready to execute applications.

A restart or restart followed by reload can be initiated either manually or automatically to recover from an error situation.

The following properties specify how the variable behaves during a start, restart, and reload.

- **CLEAR**. Variables with the CLEAR property are cleared at start and restart. Field variables are set to zero and symbol variables are set to the first value in their declaration list (which corresponds to zero). String variables are also set to zero.

- **RELOAD**. Variables with the RELOAD property are reloaded from the dump when the exchange does a restart followed by reload. The block can not change the value of these variables by itself. They can only be changed from the outside (by operator commands). Changes to the variable values are logged in command logs.

- **DUMP**. The variable value is written to a dump file before a restart. During restart the exchange prints out these values. This is done for testing and tracing purposes. The DUMP property can also later be given to variables by operator commands.

- **TRANSIENT**. The variable is unchanged during reload. Changes made to the variable are not saved in the command logs. This property is related to CP redundancy.
3.4.3 Function change properties
Blocks can be replaced or added to an exchange during run-time. This is called *function change*. When blocks are replaced, variables from the old version of the block are copied to the new version of the block. All variables, except those with the STATIC property, are copied.

- **STATIC**. When replacing a block, all variables from the old block are normally copied to the new block. In that case, all assignments made in the data sector of the new block are lost. But if the variable is declared as STATIC in the new block, the value is not copied from the old block and the variable thereby receives its value from the data sector of the new block.

3.5 Variable addressing
The RS contains a table called BAT (Base Address Table) which points out all stored data in DS. All variables must be pointed out by the BAT. Figure 5 gives a simple overview of the addressing. A more detailed description can be found in reference [1].

![FIGURE 5. Variable addressing](image)

3.6 File
A *file* in Plex does not represent the equivalent of a file in many other programming languages. Files in Plex have nothing to do with the file system. A very short explanation is to say that a *file* is an array of *struct* variables, where *struct* refers to the C programming language type constructor.
Every element in a file is called a record. Specification of which element to use is done with a pointer.

The number of records in a file is specified in the data sector. The size of a file can be declared as fixed or alterable.

The data initialization is done when blocks are initially loaded into the exchange. Exception to this is if the file size is fixed.

### 3.6.1 Record
A record collects many variables into one unit.

Variables in a record are called individual variables. DS variables that are not part of a record are called common variables. They are always stored (DS or BUFFER) and may be field, symbol, or string variables. Two-dimensional arrays are not permitted to be used as individual variables.

### 3.6.2 Pointer
To indicate a specific record in a file, a pointer is used. Pointers are record indexes that are associated to a file. The value of the pointer is the number of the current record. Many pointers can be declared for the same file.

Pointers are not temporary variables but they are stored in the register memory. This means that they behave like temporary variables and lose their values when execution leaves the block.

Pointers have a length of 16 bits as default. But they can be declared with the R property, which makes the length dependent of the register size in the target processor.

### 3.6.3 File size alteration
The file size is the number of records in a file. The file size is either fixed or alterable. All files in a certain block must either have a fixed or an alterable file size, the two different types can not be mixed within a block.

When using fixed file sizes, the file size is initialized once and can not be changed.

When using alterable file sizes, the file size is initialized with the minimum number of needed records, which later can be increased or decreased.
An alterable file size makes better use of the memory, but also gives overhead. For this reason, alterable file sizes are not recommended to be used unless necessary.

3.7 Signals
The blocks interact with signals. Sending a signal basically means that the execution does a direct or delayed jump. A signal can be sent to its own block or to another block.

When sending a signal, the sender uses a table called Global Signal Distribution Table (GSDT) to find out where to send the signal. This table is used to get the Block Number Receiving (BN-R) and the internal address of the signal.

At the startup or restart of an application system, the first signal sent is an initialization signal. The main purpose of this signal is to execute some data initializing code.

There are different types of signals, depending on where the signal destination is located. When sending a signal, a number of properties are used. A signal is able to carry data.

3.8 Signal data
Signal data are variable values sent with a signal. Signal data may consist of field variables, symbol variables, string variables, pointers, and BUFFER variables.

Signal data can only be received as field variables, string variables, pointers, and BUFFER variables.

Symbol variables can be sent, but they cannot be received as symbol variables. All symbol values translate into their associated value, so the signal only carries a numeric value. A good way of sending symbol variables is to translate them into local number symbols before sending them.

3.9 Signal properties
There exist three groups of signal properties. Each CP-CP signal (CP-CP signals are explained in Section 3.10) has one property from each of the groups. Below follows a description of the properties in each group.
The combined and buffered properties can not be used together. All other combinations are possible. This gives totally six different kinds of CP-CP signals.

### 3.9.1 Unique / Multiple
A signal is either *unique* or *multiple*. This property decides if a signal can be received by one or many blocks (not at the same time). In Figure 6, both cases show a signal that is sent to Block B. But in the case of multiple signals, the signal could have been sent to Block C instead of Block B. This is not possible in the case with the single signal.

A **unique** signal can only be received in one particular block, which is specified when declaring the signal.

A **multiple** signal can be received by any block. Therefore, the receiving block needs to be specified when sending a multiple signal.

![Diagram of unique and multiple signals](image)

**FIGURE 6.** Difference of unique and multiple signals

### 3.9.2 Direct / Buffered
A signal is either *direct* or *buffered*. This property decides if a signal should behave as a direct or delayed jump. Figure 7 shows that only the buffered signals are delayed.

A **direct** signal is sent immediately. It does not pass through any of the job buffers. Direct signals are similar to direct jumps to another block, i.e. the following statements are never executed. Direct signals makes the execution continue without any possibility of interrupts, all other signals have to wait.
A buffered signal is queued in one of the job buffers before it reaches the receiving block. It is not predictable when the signal reaches the receiving block. Buffered signals are similar to delayed jumps. The execution is not normally interrupted when a buffered signal is sent.

![Direct signal](image1)

**FIGURE 7.** Difference of direct and buffered signals

### 3.9.3 Single / Combined

A signal is either single or combined. This property decides if the sent signal is followed by a reply or not.

A **single** signal is not followed by a reply.

When a block sends a **combined** signal, it first sends a combined forward signal. Then the receiving block sends a reply (called combined backward signal). Combined signals always implicitly have the direct property, this is the reason why the combined property can not be used together with the buffer property.

![Single signal](image2)

**FIGURE 8.** Difference of single and combined signals


3.10 Signal types

The CP can send and receive both external and internal signals. This is illustrated in Figure 9. The external signals are sent to or received from the RPs and are called CP-RP and RP-CP signals respectively. Both CP-RP and RP-CP are often called RP signals. The internal signals are sent and received by the CP itself and are called CP-CP signals.

This thesis mainly concentrates on CP-CP signals. All above described signal concepts hold for CP-CP signals, but not necessarily for RP signals.

![Figure 9](image.png)

FIGURE 9. The different types of software signals

3.10.1 CP-CP signals

CP-CP signals are sent from one block to another. CP-CP signals can have the different properties mentioned above.

3.10.2 RP signals

All signal descriptions above mainly apply only to CP-CP signals.

RP signals do not have any of the ordinary CP-CP signal properties. They behave like unique, buffered, and single signals.

RP signals are either CP-RP signals or RP-CP signals.

CP-RP signals queue in a separate job buffer called JBR (Job Buffer R). RP-CP signals queue in one of the four job buffers JBA, JBB, JBC, or JBD.

The CP contains a table called the distributed RP table. This table contains the physical addresses to all RPs. The addresses are used by CP-RP signals to find the right destination RP along the RP bus.

RP signals can not carry signal data that are string variables or BUFFER variables.
3.11 Signal addressing

Here follows a description of how a signal is sent from one block to another. Figure 10 gives an overview over the procedure.

Every block in PS contains a table called Signal Distribution Table (SDT). The table has an entry for every signal that can be received by the block. The entry for a certain signal contains the Instruction Address (IA) to where the execution will start when the corresponding signal is sent to the block.

Every signal that can be sent to a block is identified in the block by a Local Signal Number (LSN). The Local Signal Number is an index into the Signal Distribution Table.

Together with the receiving block number, the Local Signal Number is used to get the Instruction Address of the signal handler. The receiving block number is called Block Number Receiving (BN-R).

To keep track of all signals in the application, there exists a table called Global Signal Distribution Table (GSDT). Every signal has an own entry in this table. Every entry contains the Block Number Receiving and Local Signal Number for the associated signal.

Every signal in the application has a unique Global Signal Number (GSN) which is an index into the Global Signal Distribution Table.
Every block in PS also contains a table called Signal Sending Table (SST). Every signal that can be sent from the block has an entry in the Signal Sending Table. An entry in the Signal Sending Table contains the Global Signal Number for the associated signal. The signal sending statement includes a Signal Sending Pointer (SSP) that is an index into the Signal Sending Table for the signal that is to be sent.

To send a signal from a block, the first thing to do is to use the Signal Sending Pointer as an index to get the Global Signal Number from the Signal Sending Table. The Global Signal Number is then used as index into the Global Signal Distribution Table to get the Block Number Receiving and the Local Signal Number. Then the Local Signal Number is used in the Signal Distribution Table of the receiving block to get the Instruction Address for the signal. The Instruction Address is then used to start the execution of the signal handler.

Only the most necessary information is provided in Figure 10. The following things are omitted and will not be further explained (the interested reader can find more information about them in references [1] and [19]).

- There are actually two different types of Global Signal Distribution Tables. One for unique signals called Global Signal Distribution Table Unique (GSDT-U) and one for multiple signals called Global Signal Distribution Table Multiple (GSDT-M).
- The reference table is used to get an absolute block address from the Block Number Receiving.

A signal with the direct property is shown in Figure 10. A signal with the buffer property would be delayed after receiving the Block Number Receiving and the Local Signal Number. It would save these and the signal data into the JBUM. After queuing it will continue.

### 3.12 Job handling

A job is a continuous sequence of statements executed in the processor. A job starts when a subfunction is called and ends when the control is given back to the job handler. This means that a job can include direct and combined signals, and are therefore not limited to one block.
A *subfunction* starts with a signal-receiving statement and exits by leaving the control back to the jobhandler or by sending a direct signal. This means that the subfunction is limited to one block.

A *signal* is used to start the execution of a job. The job starts when the signal is sent.

The CP can only execute one job at a time. Therefore, all signals but those with the *direct* property are either placed in a job buffer, the job table, or a time queue. These concepts are further explained in Sections 3.12.1, 3.12.2, and 3.12.3 respectively. The job buffers, job table, and the time queues do not store *jobs*, they store *signals*.

Jobs can be assigned to five different priority levels. Listed with the highest priority first these priority levels are: *Job Table*, *JBA (Job Buffer A)*, *JBB (Job Buffer B)*, *JBC (Job Buffer C)*, and *JBD (Job Buffer D)*.

The priority levels can be further divided into three different groups. Jobs within the same group can not interrupt each other. The Job table, JBA, and JBB belongs to the *THL (Traffic-handling level)*. The JBC belongs to the *BAL 1 (Base level 1)*. The JBD belongs to the *BAL 2 (Base level 2)*. The priority levels and their associated groups are shown in Figure 11.

![FIGURE 11. The different priority levels of jobs](image)

Every 10 milliseconds, a CIS (Clock Interrupt Signal) is sent. If no job from the job table, JBA or JBB is executing, scanning of the job table is started. Scanning of the job table is the process of searching it for signals that are to be sent. After that, the JBA is checked for signals, then the JBB. If no job from JBC is executing, JBC is checked for signals. If no job from JBD is executing, JBD is checked for signals.

If a new job enters an empty job buffer, the buffer sends an interrupt signal for that priority level. If the executing job belongs to a group that has lower priority than the
group of the new job, the executing job is interrupted. This means that the system uses preemptive scheduling.

Figure 12 shows an overview of the job handling of the job table and job buffers. Every interrupt takes control of the processor. The cases where the interrupt returns the control are not explicitly drawn in the figure. Every choice that is not drawn in the figure corresponds to returning the control.
An example of job handling is shown in Figure 13. From the beginning, no job is executing. When the CIS arrives, the job table starts scanning. During scanning two interrupts arrive. After the job table has finished scanning, the B-level job starts
executing since it has higher priority than the D-level job. After the B-level job has finished executing the D-level job starts executing.

When the D-level job is executing an A-level interrupt arrives. Since the A-level job belongs to a group with higher priority than the group of the D-level job, the execution is interrupted and the A-level job instead starts executing. During execution of the A-level job a C-level interrupt arrives. When the execution of the A-level job has finished, the C-level job starts executing. The D-level job has to wait since the C-level job has higher priority.

When the C-level job is executing a B-level interrupt arrives. The C-level job is immediately interrupted. During execution of the B-level job a CIS interrupt arrives. Since the job table and the B-level job both belong to the THL group, the Job table has to wait until the B-level job has finished executing.

When the B-level job has finished executing, the Job table starts scanning. After the Job table has finished the scanning, the C-level job continues its execution. When the C-level job has finished its execution, the D-level job finally gets the chance to finish its execution.
It is possible to disable the interrupt feature from within a subfunction. When the interrupt feature is disabled, only code in the subfunction is allowed to execute. It is not possible to send combined signals or in any other way exit the subfunction. Sending buffered signals without exiting is permitted.

In APZ, lower level jobs are immediately interrupted. However, in APZ VM ([6]) small delays occur before the lower level jobs are interrupted. The lower level jobs periodically check if they should be interrupted. Small jobs might not even become interrupted. A special case occurs when a block sends a buffered signal with a higher priority. In that case no delay is allowed. The reason is that an application can be designed on the premise that the interrupt will occur immediately.
3.12.1 Job buffers
There are totally five job buffers. Four of them are for CP-CP and RP-CP signals, and the last one is for CP-RP signals.

The job buffers belonging to CP-CP and RP-CP signals are JBA, JBB, JBC, and JBD. The buffers are ordered according to their associated interrupt priority, where JBA has the highest priority.

The job buffer belonging to CP-RP signals is JBR. The CP sends these signals to the RPs.

The approximately maximum number of signals in each of the queues for APZ are: JBA - 512, JBB - 4096, JBC - 4096, JBD - 512. The exact number depends on the APZ version.

3.12.2 Job table
The job table is used for jobs executed at short periodic intervals.

For each signal placed in the job table, there is a corresponding counter. The counter is proportional to the delay time before the signal is to be sent. Every 10 milliseconds (i.e. at every CIS), the job table is scanned. All counters are decremented by one (unless the counters equal 0 or #FFFF). When a counter reaches zero, the corresponding signal is sent.

The maximum value for a counter in the job table is #FFFE.

There are procedures in Plex for inserting signals into the job table. There are also procedures for removing signals from the job table and for changing the counter for signals in the table. The last mentioned procedures can only be used when the signal has been sent.

When a signal in the job table has been sent it is not automatically deleted. It instead becomes inactive. It is then up to the programmer of the signal handler to decide if the signal is to be removed or if the counter is to be updated.

Only CP-CP signals can be inserted into the Job table. The signals must not contain any data.
When inserting a signal into the job table, the time after which it is to be sent is specified as the number of 10 ms time periods.

3.12.3 Time queues
Signals can be delayed using time queues. Time queues delay buffered signals in longer intervals than the job table. There is one absolute time queue called TQA (Time Queue A) and three relative time queues called TQB (Time Queue B), TQC (Time Queue C), and TQD (Time Queue D). A signal always moves from the time queue to one of the four job buffers. This is illustrated in Figure 14.

The absolute time queue stores absolute times for signal execution. The absolute time is specified in month, day, hour, and minute. Every minute, the time queue compares this value to the system calendar. When there is a match, the signal moves to one of the four job buffers.

According to reference [2] the signals can be set to repeat in the absolute time queue. By setting the day number to zero, the signal will be sent every time the hour and minute matches the current time (i.e. once a day). By setting the month number to zero, the signal will be sent every time the day, hour, and minute matches the current time (i.e. once every month that has that particular day).

The three relative time queues have a counter for each job. Every 100 milliseconds, 1 second, and 1 minute, respectively, the time queues receive a periodic signal from the job table and decrement the job counters. This is shown in Figure 15. If a counter reaches zero, the time queue forwards the corresponding signal to one of the job buffers. The maximum counter value is #FFFE.

Every time queue in APZ is able to store 512 signals.
FIGURE 15. Overview time queues
4 Solution

This chapter describes the solution to the problem that was to be solved. There are two aspects of the solution. One is the construction of a run-time environment and the other is the specification of how a compiler is to compile code to match the run-time environment.

Not all compiling directives need to be followed, since some of them are not depending on the run-time environment. These can instead be seen as tips for the compiler developer. They are included in this thesis since they had to be examined during the thesis work. There was no need to include them in the run-time environment.

The design in the chapter is illustrated as Java source code. A compiler will however not translate Plex to Java, but Plex to Bytecode. Using Bytecode enables more options to optimize. But it would be very complicated and would not fit into the scope of this thesis. Bytecode optimizations would probably not have much noticeable effect and they might destroy optimizations performed by the JVM. Therefore Bytecode is not considered in this thesis.

Since most constructs in Plex are static, many things can be checked during compilation.

The theory behind the decisions is in most cases only explained in Chapters 2 and 3. Consider those chapters for more background information.

The run-time environment was implemented and works under Windows NT and Tru64. All testing was done on Compaq Alpha Tru64 since that is the primary target platform.

Different versions of the APZ places different things in hardware and software. But it does not matter which are in hardware or software, since the run-time environment must model both of them in software.

4.1 Blocks

All blocks must be compiled into a format that is specified by the run-time environment. The block design is modelled as a Java class. The real compiler will
Solution

translate this to Bytecode instead of Java. To translate Plex blocks to Java classes is a very natural approach. With the help of the Java classloader, it is also a very good way to enable the system to do function changes.

The biggest question is how to design the blocks so that they can receive signals. Both the block and signal design must be considered carefully. The signals must have the ability to be queued. After a signal has passed through the queue it is to be sent. It must contain some information on how to find the destination to where in the system it should be sent. A queued signal must also contain its signal data.

The only way to make the execution jump to another class in Java is to use method calls. Therefore the destination must be a method combined with an object. This corresponds to a signal handler in a specified block in Plex.

The destination of the signal must somehow be saved into the queue. An object reference is easily stored, but not which method that should be called. For unique signals this could be hard coded into some signal object, but this is not an alternative with multiple signals since not all destinations are known for them. That approach is therefore not possible.

Function pointers seem to be a good solution to the problem. That way a method reference and an object reference could be stored into the queue. But function pointers are not available in Java, in contrast to for example the C programming language.

Function pointers can however to some extent be modelled in Java. The problem is solved by creating a signal object which contains all the parameters (i.e. the signal data) and a reference to the object it is to be sent to. The object implements a signal interface. The interface contains a method for sending the signal. That way the signal could be placed into the queue and then later be sent when the run-time environment wants it to. When the signal is sent the corresponding destination object can call the right method depending on which signal object it has received.

Some inspiration to this solution comes from reference [20]. The solution might be easier to understand by studying Figure 18 and Figure 19.

Another way to solve the problem is to use Java’s reflection mechanism. That way the correct method can be invoked from the queue. But reflection is slow and would
make a slow solution. This and other disadvantages of *reflection* are stated in reference [10] (Item 35).

As stated above, the queue needs to know where to send the signal. The reference to the signal receiver is placed within the signal. This is shown in Figure 16.

The reference to the signal receiver could either be an index into a table where the real reference is or it can be the real reference. The approach with the table makes it easier to do a *function change* (since the only reference to the signal receiver is placed in the table), but then the compiler needs to generate this table. To make it easy, this design does not use a table. There is also not any need for supporting function change at the moment. A table can be inserted later with minor modifications. The table approach is shown in Figure 17.

The signal receiver objects could either be created during system initialization or they can be created when they are needed. Creating them directly increases speed but allocates memory for all the receivers (even if they are not used). It also needs the compiler to generate code for creating the objects. This design uses the easiest approach of creating the objects when they are needed. This task is made really simple by using the *Singleton* design pattern. A Singleton design pattern handles the global reference to the only instance of the corresponding class that can be created. More about the Singleton pattern can be read in [8] and [10] (Item 2).
The three relevant parts of a block are the declare, program, and data sectors. The ID sector is not relevant since it only consists of comments and will not be compiled anyway.

The declare sector is used to declare all variables and constants. Blocks in Plex are static. This design does not use static classes for several reasons. Since a reference then would not exist, there would be problems with queuing signals and function changes. Reflection could be used to solve that problem, but as stated earlier that is not a good idea. Static classes also make an ugly design. But since only one instance of each class is created, the variables and constants can be modelled as instance variables. They are available to all signals that are received within that block. More about variables will be explained later in this chapter.

The program sector contains all signal-receiving statements. These are modelled as methods. All these methods take the sent signal as their only argument. The queues must be able to store all kinds of signals. This is done by letting all signals implement a signal interface. When they later are sent to the receiving block, the block needs to check what signal type it is and then call the appropriate method.

The data sector contains the initial values for the variables. This is solved by initializing the variables when they are declared. The initialization takes place when the class is loaded.

Every block in Plex is able to receive signals. To be able to receive signals, the blocks must implement an interface called Block. This interface contains the enter()-method, which all blocks must contain. This greatly simplifies the sending of signals, since it now can be done with the use of polymorphism. The block has a method that redirects all incoming signals to their corresponding method.

As mentioned above, all blocks are implemented as Singletons. Every time a block is accessed it is done through the static variable with the same name as the block. When this variable is accessed for the first time, the block instance is automatically created. This instance is not deallocated later when the garbage collector has run. The singleton instance is always referenced by its corresponding class, so it will not be collected even if no other references exist.

Figure 18 shows the Java design of a block, including two example attributes and two example signals. Package and import directives are omitted from the figure.
Many parts of the block design are explained later in this chapter. The code with bold text style is different for different blocks.

A bad thing with the block design is the extensive use of the instanceOf-operator. The instanceOf-operator is slow. It would be better if the overloaded enter()-methods were called automatically at run-time. However, this is not possible since the choice of which overloaded method to invoke is done at compile time. This is different to the above used overriding, where the choice is done at run-time.

If the enter() method is called with a signal that the block can not receive, an java.lang.IllegalArgumentException ([15]) is thrown. This exception is just to help the designer of the compiler to spot errors. During compilation the compiler should check that no signals are sent to blocks that can not receive them. If this happens, a compilation error occurs.
4.2 Addressing

Because of Java’s approach to handle references and classes, all stores can be eliminated.

The run-time environment does not need to implement the DS since all the advantages with the DS are provided automatically by the JVM. The JVM has built-in mechanisms for taking care of memory protection, name spaces, and loading and unloading of classes.

The question about variables is how to represent them.
4.3 Variable types

This section first gives a short abstract of the variable types in Java. This is followed by a description of which Java variable types that will be used for the different Plex data types. More information about Java data types can be found in reference [13].

4.3.1 Java variable types

The Java programming language has two kinds of data types: primitive types and reference types.

The primitive types can be divided into numeric types and the boolean type. The numeric type can further be divided into integral types and floating point types. The integral types consist of byte, short, int, long, and char. The floating point types are of no interest here, since Plex does not support that kind of variable types. Table 2 shows all the relevant Java primitive data types.

The reference types are class, interface, and array. The array type is the only of these that is relevant for variable choice. But the class type is used for files in Plex, which are explained in Section 4.5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Min value</th>
<th>Max value</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128</td>
<td>127</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32768</td>
<td>32767</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td>16 bits</td>
<td>0</td>
<td>65535</td>
<td>0</td>
</tr>
</tbody>
</table>

The use of the Java primitive types is faster than using own declared types. Because of that, the goal is to translate all Plex variable types to Java primitive types.

The suitable primitive types often have a greater range than the corresponding Plex variable. This can give problems depending on how programmers use the variables. If they write code that depends on the variable range, the range needs to be simulated. An example of such code is a loop that continues until the variable gets out of range and wraps around to zero again.

The simulation can either be done in the run-time environment or by the compiler. If it is done in the run-time environment, this will need declaration of own types (by
using classes for types). This would give large overhead. If it instead is done in the compiler, the compiler can generate suitable code that checks for range. This would give less overhead. The compiler is also able to do optimizations to lower the number of times the range needs to be checked.

If the checks are placed in the run-time environment, there will be less memory needed for extra checking code. If the checks are placed by the compiler the code will be faster. In this system the last alternative is most suitable, since high speed is preferred to low memory usage. Since the checks are so small, the difference in memory usage normally will not be big anyway.

The \texttt{int} Java datatype is the natural size used by the JVM ([30]). When operations are performed with \texttt{byte}s, \texttt{short}s, and \texttt{char}s they are performed using \texttt{int}s. Assigning the result to a variable of one of those types also requires an explicit cast. This cast generates an extra Bytecode instruction. This makes it seem like the performance actually becomes worse when using smaller data types. But this does not hold when using a JIT compiler. The JIT compiler uses those data types that are handled effectively by the processor, it does not have to use \texttt{int}s. The run-time environment uses a JIT compiler, it is therefore important to choose suitable data types.

### 4.3.2 Field variables

The primitive types in Java do not fit nicely to the field variables. The variables that will be used are shown in Table 3.

The boolean type can not be used at all since it does not support the arithmetic operations and some of the comparison operators. The reason for this is that Java does not treat boolean as a type that corresponds to an unsigned integer. Many other languages, for example the C programming language, do this and there is \texttt{false} equal to zero and \texttt{true} is equal to everything else. This does not apply in Java.

Most of the Java types are \textit{signed}. This is a big problem since all field variables in Plex are \textit{unsigned}. This is fine as long as the types in Plex have a length that is shorter than the types in Java. When the length is equal, translations could be done. This is possible, since the number of different values is the same. However, many problems arise. A lot of conversions between the two different ranges would need to be done. This would cause overhead. Instead types that have longer length are used. This will also give some overhead since a larger variable type is used, but that overhead will not be as large as in the other approach.
One exception to the above mentioned problem exists. Java has one unsigned integral type, the \textit{char} type. This is a fact that is very good. The char type is 16 bits long, that corresponds perfectly to the 16 bit field variables that are very much used in Plex. It may seem strange to use a type made for characters as a container for integers. But it does not matter, since the char type is as much integral type as the other integral types. It thereby supports the same operations.

The Java type for R variables is specified by the used processor. There are only two choices. 16-bit R variables use the char type and 32-bit R variables sadly need to use the long type (because of the above mentioned problem). If 64-bit R variables existed they also would have to use the long type, but that makes the need for above mentioned conversions. These conversions are done by the compiler. There also is another solution, the usage of \texttt{java.math.BigInteger} (the specification can be found in reference [15]). These objects can represent any integer and thereby suit any processor. Though the construction of an own object will probably be more effective. But in any case, it is very bad for performance to use objects for each of these kind of variables.

Arrays correspond well to the Array reference type in Java. When an array is declared in Plex, both the array and the elements contained in the array are implicitly declared. This is not the case in Java, so this needs to be done explicitly.

\begin{table}[h]
\centering
\caption{Implementation of Plex field variables in Java}
\begin{tabular}{|l|l|}
\hline
Plex & Java \\
\hline
Field 1 & byte \\
Field 2 & byte \\
Field 4 & byte \\
Field 8 & char \\
Field 16 & char \\
R 16 & char \\
R 32 & long \\
R 64 & long / Object \\
\hline
\end{tabular}
\end{table}

\subsection*{4.3.3 Symbol variables}
Symbol variables only make work for the compiler. Symbol variables are normally replaced with numbers during code generation, this can be done in the same way for this system.

One optimization that can be done by the compiler is to not always use 16 bit length (which gives a maximum value of 65535) for the variables. The compiler can see
from the declaration how many symbols are used for the variable. This can be used together with the column of max values in Table 2, to decide which variable type to be used. Max values, and not the number of different values, need to be used since the variables need to correspond to the same number now as they would have done when compiled with the ordinary Plex compiler, i.e. start at zero and count upwards.

4.3.4 String variables
One approach is to exchange string variables against ordinary strings in Java. Then the program will use Unicode characters. Unicode is a character encoding standard independent of platform, program, and language.

This will work nicely for most applications. But if the application uses code that depends on the character encoding normally used in Plex, errors will occur. Because of this, the same character encoding must be used in this system. For example: “£” corresponds to number 35 in CCITT No. 5, while it corresponds to number 163 in Unicode. If an application uses the number 35 to represent “£”, an error is made.

It also saves memory not to use Unicode. Unicode needs 16 bits for every character, while CCITT No. 5 only needs 7 bits.

The solution is to create a string using an array of bytes. This suits very well since bytes make use of 7 bits to positive integers. When declaring strings in Plex the maximum length is specified. This converts to the array length in Java.

When the compiler generates code for string constants, it must use the CCITT No. 5 encoding.

4.4 Variable properties
The different variable properties in Plex give some additions to the block model showed earlier in this chapter and will thereby explain the for the moment unexplained parts of Figure 18.

An explanation of how the different Plex variable types will be compiled follows.

The variables will as far as possible be declared with private access. The RELOAD variables are for simplicity made public. This does not really matter that much, since the data encapsulation is preferably controlled by the compiler anyway.
4.4.1 TEMPORARY

The optimization of TEMPORARY variables is important, since they are very fast in Plex. If they become too slow, some Plex applications that make extensive use of temporary variables might suffer much in performance.

Temporary variables can in most cases be implemented as local method variables. The temporary variables could of course be implemented as instance variables, but to implement them as local method variables instead of instance variables gives benefits similar to using temporary variables in Plex. Since the variables then are stack variables, the access to the variables will be faster ([7] praxis 35). This is also the case with temporary variables in Plex, since they are stored in the RM. Accessing stack variables are generally faster than accessing class or instance variables since the JVM is a stack-based machine. Since the JVM stack is virtual, this is to some extent dependent of the JVM implementation.

To implement variables as local method variables has additional benefits. If they were implemented as instance variables, the same instance of the variable would be accessible from every signal handler in the block. This would be a problem, since more than one signal handler can be executing at once. For example, a C-level job (belongs to group BAL1) may execute and use a temporary variable. Suddenly, it might get interrupted and an A-level job (belongs to group THL) may use the same temporary variable. This can not occur in Plex, since the different job group levels use different temporary registers. If the A-level job changes the value of the variable, it corrupts the variable value for the C-level job. This would make it necessary to have one separate instance variable for each priority group level where the temporary variable is used. This problem does not occur when using local method variables. Then the variables for different methods will not be shared, since they are created on the stack for each method call.

There are some exceptions where instance variables must be used. All these exceptions can only occur when the temporary variables are used as input or output parameters in subroutines. Plex does not have any input or output parameters ([1]). Since Plex does not use return values, subroutines “return” results by placing them into temporary variables. Since Plex does not either use parameters when calling a subroutine, parameter values may be assigned to temporary variables and then the subroutine is “applied” on them.
In many cases an ordinary Java method call can be used to call the subroutine with the temporary parameters as method parameters. The return value from the method can be used for one Plex out-parameter. If a subroutine uses more than one return value, only one of the return values can be returned by an ordinary Java method call. The other return values have to be returned to the caller by using instance variables. Another solution could be to create objects that contain all the return values, but this would cause unnecessary overhead. This may not either really be an optimization since the parameters instead need to be copied during the method call.

For the temporary variables that must be instance variables, the above stated problem will hold. Those temporary variables might be used by more than one method running in different priority group levels (THL, BAL1, or BAL2). This means that the same problem stated above can occur. The solution to this is to store separate versions of those temporary variables for each different group priority level they are used in. A parameter, that specifies which parameters to use, also needs to be added to each subroutine.

It may be hard to know which variables in a subroutine that really are input or output parameters since this only is indicated by comments in the code. These comments can not be trusted by a compiler. At least those variables that only are used in that subroutine (not in any other subroutine or signal handler) are *neither* input nor output variables.

Since TEMPORARY variables always have a length of 16 bits, they are always converted to char variables.

**4.4.2 DS**

DS variables are very similar to instance variables in Java and are therefore modelled as such. They can not be modelled as local method variables as in the case of temporary variables. This is because their values must be saved between different calls into the block.

**4.4.3 BUFFER**

Buffer variables are very similar to object variables in Java. Different kinds of object variables are used to implement them.

Buffer variables can be accessed by other blocks, but only when the block has a reference to them. Buffer variables are declared as private.
All buffer variables must be reference types, otherwise the receiving block will not get access to the same data (only to data with the same value). This is solved automatically since all buffer variables in Plex are integer arrays, and arrays in Java are reference types.

The datatype used for each element in the array are taken from any of the field variables in Table 3, except the R variables which are not allowed.

4.4.4 CLEAR
CLEAR declared variables should be cleared at start and restart. To make this possible, every block provides a clear() method. This method is generated by the compiler. It contains clear statements for all variables that have the CLEAR property.

4.4.5 RELOAD
These variables should be reloaded from the system dump during a reload with restart. This is implemented by every block by providing a reload() method. The method takes a java.io.FileInputStream as input (specification in [15]). The method reads the variable values from the file and assigns them to the appropriate variable. It is important to understand that the dump file used is not the same as used by variables with the DUMP property.

Since these variables needs to be able to be changed from outside the block, they are made public accessible.

The RELOAD variables should not be able to be changed from their own block. This is checked by the compiler during compilation.

When the value of RELOAD variables is changed, the change has to be logged in a command log. Code for this is to be generated by the compiler.

The block could provide getters and setters. This would mean that changes to the variables easily are logged inside the block. But for simplicity they are currently made public.

4.4.6 DUMP
DUMP declared variables should be written to a dump file. This is implemented by every block by providing a dump() method. The method takes a
java.io.FileOutputStream (specification in [15]) as input. The method writes the values of all DUMP declared variables to the file (appending to the end of file).

Since the DUMP property can be changed during run-time, some more things need to be added. The exchange operator changes a variable in a block by specifying the variable name and status. An extra boolean flag for the dump status of each variable is needed. Also an extra String attribute can be used to store the name of each variable, or reflection can be used. Reflection is not used for this since the other approach gives better time performance.

A method with the signature \( \text{boolean setDump(String name, boolean status)} \) is added to each block. This method is used when the operator sets or clears the DUMP status of the variable. The name of the variable and the status of the DUMP flag need to be specified. The method returns \( true \) if the variable existed, otherwise \( false \).

One way to implement the dump() method is to use a series of if-statements. Every if-statement uses the extra status variable to check the status of one variable, and dumps it if it has the DUMP property. This method will take some time to perform since it must contain I/O statements for writing to a file.

### 4.4.7 TRANSIENT
This property is easily implemented, since it is not supposed to do anything when CP redundancy is not considered. The variable must \( not \) be part of the clear() method or the reload() method.

This means that the compiler must check that variables that have the TRANSIENT property do \( not \) also have the CLEAR or RELOAD property.

### 4.4.8 STATIC
Those variables that have the STATIC property must \( not \) be copied to the new block. Nothing needs to be done to achieve this.

The variables that do \( not \) have the STATIC property need to be used for initializing the new block.

To make this possible, all new blocks must implement a change() method. This method initializes the new block with all non-STATIC declared variables. All other
variables are not changed (thereby saving the original value from the new block). The change() method takes all non-STATIC values as parameters.

The change() method is not part of the signal interface, since not all blocks must implement it and it does not have a standard method signature.

Function change is done by the run-time environment. To initiate a function change the blocks need to implement a method called `functionChange(Block newBlock)` that takes a reference to the new block as a parameter. The method simply calls the `change()` method in the new block with all non-STATIC variables. The `functionChange()` method is part of the block interface.

### 4.5 Files

Records are very similar to the C construct called `struct`. This construct is not included in Java, but is in this case preferably modelled as a `private static member class`. If the `static` modifier is not included, each instance will contain an extra reference to the enclosing object. In this case this has no benefits and costs both space and time (Item 18 in [10]). Since the records are private to the class, there is no real need for getters and setters, the variables are instead declared with `package` access and getters and setters are omitted.

A file resembles an array of structs. To make file size alteration possible the array needs to be resizable. A first approach could be to choose the `java.util.Vector` class (reference [15]), since it is resizable. But since the file is only accessed by one thread and therefore do not need any synchronization, a much faster choice is the `java.util.ArrayList` class (reference [15]) (see also praxis 34 in reference [7]).

Since a record `pointer` behave as a temporary variable, it is declared exactly as the other TEMPORARY variables (see Section 4.4.1).

The limitation that two-dimensional arrays not are permitted as individual variables is left to the compiler to check.

### 4.6 Signals

Signals do not actually contain anything more than their data and a reference to the block they will be received in. This gives a small signal design, which is very
important in the case of signals because additional signal objects are created after initialization of the system.

Every signal must implement an interface called `Signal`. This is necessary for being able to use polymorphism when sending the signals from queues. It also makes it easier to extend the signal design later and gives a cleaner design.

The `send()`-method uses dynamic binding and polymorphism to send the signal to the right block. This makes the implementation and code generation easier. A bad thing is that a signal can be created to be received by any block, even if that block does not support receiving of that kind of signal. The same thing is also possible in Plex. If a signal is sent to a block that does not support it, an `IllegalArgumentException` ([15]) is thrown. It would however be better if an exception occurred before the signal is sent. Where possible, this is solved by letting the compiler during compilation check that only signals with valid destinations are created. The only time this is impossible to do is when the destination is specified with a variable. But all possible destinations used by a block are known during compile time, which makes it possible for the compiler to generate code that checks this too. This can be done either by inlining code or by creating a method that performs the checks. Another approach could be to have multiple overloaded constructors, one constructor for every signal receiver that is allowed. But this is not possible since the number of receivers is dynamic and more can be added later. Reflection is also a possibility, but bad for performance. Compiler checking is the far best choice here.

Figure 19 shows the Java design of a signal, including two example attributes. As in the case with blocks, the real compilation process produces Bytecode instead of Java.

The signal data must be available to the receiving block. Therefore they are not declared as `private`. Getters could be used, but is not necessary since the code will be generated and checked by the compiler.

Both the data and methods could be declared with `package` access, but `public` is chosen to make it able to place signals and blocks in different packages. This is also done for the constructor to make it possible to place signals and blocks in other packages than the run-time environment.
Object creation is expensive (the reason for this can be read in praxis 32 of [7]). After the system is initialized, the object creation is therefore kept at a minimum. Signal objects are almost the only objects that are created. These objects need to be created, as the signal data must be stored somewhere when the signal is placed into a queue. To make these objects available for garbage collection as soon as possible, the references are set to null immediately when they are no longer needed (see also praxis 7 in [7]). Another approach could be to use preallocated arrays in the queues for storing signal data. This would however later make a need of reflection when the signal should be sent, it would also suffer in clarity of the design.

One idea to make the memory management for constructing all signal objects more efficient could be to use an object pool. An object pool is a data structure that contains allocated objects that are reused instead of reallocated. But reference [9] states that this is a bad thing. Object pools fool the garbage collector that objects are live, even though they are not. Object pools could have worked before exact garbage collection was used, but it does not work in any modern JVM. Also reference [10] (Item 4) states that an object pool is a bad idea if the objects are not extremely heavyweight. Object pools increase memory footprint and harm performance. A modern garbage collector does easily outperform object pools on lightweight

```java
class Signalname implements Signal {
    // Signal receiving block.
    private final Block receiver;

    // Signal data.
    public final datatype1 attributename1;
    public final datatype2 attributename2;
    ...

    public Signalname(Block receiver,
        datatype1 attributename1,
        datatype2 attributename2,
        ...)
    {
        this.receiver = receiver;
        this.attributename1 = attributename1;
        this.attributename2 = attributename2;
        ...
    }

    public void send()
    { receiver.enter(this); }
}
```

FIGURE 19. Signal design in Java
objects. The signal objects are very lightweight. The cost of creating objects in a modern JVM is not as large as it has been earlier.

### 4.7 Job handling

Signals can be delayed by using either the job table, one of the job buffers, or one of the time queues. If any of the time queues are used, this is always followed by delay in a job buffer.

The signals are stored in two different ways. Both the job table and the time queues store signals in tables. The job buffers store signals in queues.

Below follows a description of the different data structures that are used for storing signals.

#### 4.7.1 Job buffers

The job buffers are implemented as *circular queues*. This gives them good performance since they are fixed-sized and rather small.

#### 4.7.2 Job table

Signals placed in the job table are most easily stored in a table. The table needs two entries for each signal, a signal reference and an integer for keeping track of the interval.

Since the table is statically allocated it can easily be implemented as a two dimensional array with two columns.

There are four different table operations that must exist:

- *insert*(signal, interval). Insert signal with the specified interval into the table.
- *scan*. Scan the table and send signals when their specified interval has passed. It does not delete signals that are sent, it just inactivates them.
- *delete*(signal). Delete signal from the table. This can only be done on inactive signals.
- *setInterval*(signal, interval). Set interval on the specified signal. The signal is thereby activated again. This can only be done on inactive signals.

The performance of the job table is very important since it is used very often. It is used as least once every 10th millisecond (in mean value).
The handling of intervals during scanning can be done in two different ways. The first way is to decrease all intervals of active signals and send (and inactivate) those signals that become zero. A better and faster approach is to keep an internal counter that is incremented during each scan. When a signal is inserted the current internal counter is added to the interval counter. During scan, all that needs to be done is to compare this interval to the current counter, and thereby there is no need for decrementing the interval. The last approach is faster since only one increment is needed during each scan. In the first approach all active signal counters must be decremented.

When the worst-case running times are specified below, \( n \) is the number of elements in the table and \( m \) is the maximum number of elements that can be placed in the table.

The probably easiest implementation of the table is to scan the whole table in each of the operations.

- The scanning of the table implies that the whole table needs to be scanned. This gives a worst-case running time of \( O(m) \).
- Inserting a signal scans the table until a free spot is found, where the signal is inserted. This gives a worst-case running time of \( O(n) \).
- Deleting and setting the interval for a signal scans the table until the signal is found, then the signal is either deleted or updated. This gives a worst-case running time of \( O(m) \).

This approach is showed to the left in Figure 20. This approach is very time consuming. Scanning needs to cover the whole table, and the other operations would have to cover more and more of the table when it contains more signals or becomes more fragmented.

Another approach is to keep all elements in the beginning of the table.

- The scanning only needs to scan until end of elements. This gives a worst-case running time of \( O(n) \).
- The signal is inserted at the end of the elements. This gives a worst-case running time of \( O(1) \).
Deleting and setting the interval scans the table until the signal is found, then the signal is either deleted or updated. When a signal is deleted, the last element takes its place in the table. This gives a worst-case running time of \(O(n)\).

This approach is more effective than the first approach. Since the table never becomes fragmented, scanning only needs to cover the parts that contain elements. The insertion operation in this approach is very effective, no scanning is needed. The approach is showed in the middle of Figure 20.

Although the second approach is better than the first, it still has a major drawback. During job table scan, all elements are covered. The scan of the job table is done very often (every 10 ms in mean value) and is very important to be executed fast. By keeping the table sorted with respect to send time, the scan can be done very fast. This approach uses three cursor implemented linked lists. One list is used to store the sorted elements. Another is used to store all free table entries. The last one is used to store all inactive signals. Using linked lists need an extra column.

The sorting is done using insertion sort. This is not a sorting algorithm with good performance, but it behaves ok since only one entry is inserted at once.

- The scanning only checks the first element in the sorted element list. If that element’s interval has passed, it is sent and then placed in the list of inactive elements. The first element is checked until no signal is sent (in case there are many signal with the same sending time). This gives a worst-case running time of \(O(n)\) (in case there are many signals with the same sending time), but often \(O(1)\).
- A signal is inserted in the first element of the list of free elements. The list of signals is then scanned for the right place to insert the signal. The list of signals is always kept sorted. This gives a worst-case running time of \(O(n)\). This can be lowered by using a better algorithm instead of insertion sort.
- Deleting an element involves scanning the list of inactive signals until the signal is found, then it is deleted. The entry is placed in the list of empty entries. This gives a worst-case running time of \(O(n)\), but often \(O(1)\).
- Updating the interval of a signal involves scanning the list of inactive signals until the signal is found, then it is updated and placed in the right place in the list of signals (in the same way as done during normal insertion). This gives a worst-case running time of \(O(n)\). This can most often be lowered by using a better algorithm instead of insertion sort.
The search through the inactive list is mostly done in a worst-case running time of $O(1)$ since it rarely contains more than one element.

This approach is the one that is used in the system. It is shown to the right in Figure 20. The scanning is done extremely fast. Inserting an element is slower than in the second approach since the list of signals needs to be scanned in order to keep it sorted. But scanning of the table is done much more often than inserting a signal. As long as the insertion time does not increase very much, this approach is the best choice. Finding the signal during deleting and updating the interval is faster since fewer elements need to be scanned (however this could also have been implemented in the second approach). But the changing of the interval has the overhead of inserting the signal in the right place in the signal list. A similar overhead also occurs in approach 2.

The insertion of entries into the inactive or free list should be done at the front (where the pointer is), this makes it a stack. Since the last inserted element in the inactive list often is the one that is used, this improves the speed. The free list does not really matter, but since there already is a pointer to the front of the free list it is most easy to use this approach.

It is hard to choose which one of the second or third approach is the best to use. The second approach can be made faster by implementing the list of inactive signals (this is preferred, since the number of interactive signals are small compared to the number of active signals) that is used by the third approach. This would make the second approach somewhat faster when inserting signals and updating signal intervals (since this includes inserting a signal). But the third approach is still much faster during scanning of the job table. This makes the third approach a more secure choice.

Since the scanning of the job table is an operation that is executed far more often then the other operations, it is very important that it is fast and efficient. But on the other hand, it is also important than inserting of signals do not take too much time.

If another algorithm instead of insertion sort is implemented, the choice of the third approach is more obvious. A very good substitution for the algorithm is a priority queue (described in reference [34]). If this approach is used, the linked list for the signals is replaced by a priority queue. The key for each element is the associated send time. There are many good implementations of priority queues available. The
Fibonacci heap (described in reference [31]) is a suitable implementation since it makes both the scanning of the job table and insertion of signals fast. The worst-case running time for scanning will be $O(n)$ (in case there are many signals with the same sending time), but often $O(1)$. The worst-case running time for insertion of signals will always be $O(1)$.

When the table is created, only the free list points at it. The free list starts at the first element and continues one element at a time until the last element is reached.

These examples can be created according to this:
Insert signals: s6, s3, s1 (15), s2 (38), s4, s5, s8, s7 (21).
Send signals: s6, s5, s3, s4, s8.
Delete signals: s3, s4, s8.

**FIGURE 20.** Different table approaches

### 4.7.3 Time queues
There are one absolute time queue and three relative time queues. The job table has control over when the different time queues will execute (as shown in Figure 15).

The data structure used for storing signals in the relative time queues is very similar to the data structure used for storing signals in the job table. Both of them need a counter for each signal that decides when the signal is to be sent. The difference is that in the relative time queues the signal entry is automatically deleted after the signal has been sent. This is not the case with the job table, where signals still are
stored in the table after they have been sent. In the job table there are commands for
removing a signal from the table and restart it by updating the counter for the signal.
The relative time queues are therefore implemented similarly as the job table but
have no inactive list.

The absolute time queue works approximately in the same way as the relative time
queues. But the absolute time queue does not have counters, it has dates. But it can
still be implemented just as the relative time queues. There is some addition for
handling of dates. There is no need for an inactive list, just as with the relative time
queues. Sent signals are not deleted, some of them are updated to be sent again next
day or month.

For each signal in the absolute time queue, a date must be stored. The date could be
stored as a java.util.Date object ([15]), but it is instead stored in milliseconds. This is
much more effective. This can be done, since every date has an associated time that
is measured in number of milliseconds since January 1, 1970, 00:00:00 GMT. This
only needs a variable of type long, instead of a whole object for every signal.

Although the time is stored in milliseconds, still an object for handling dates is
needed. The object that is used for this is an instance of java.util.GregorianCalendar
([15]). This object handles conversions to milliseconds when queuing a signal (since
the date is specified as a date, not a number of milliseconds), calculating next day or
month, and checks that specified dates are valid.

The current time in milliseconds is received directly from the java.lang.System class
([15]).

When queuing the signal, the date needs to be specified. This can either be done
using an own designed object, a java.util.Date (bad since the getters and setters are
deprecated), a java.util.GregorianCalendar, or just simple primitive variables.

The simple variables were chosen. One reason for this was to avoid instantiation of
an extra object. The instantiation of an extra object is expensive in the case with
GregorianCalendar, but an own designed object will not be that expensive. These
objects are not created so often and the main reason for using primitive variables
was to eliminate the need for an extra public class. The interface to the run-time
environment is preferably kept simple and not cluttered. The bad thing with using
primitive variables is that the parameterlist in the method becomes very long and
hard to remember. It can easily be misused since the compiler will not notice if any of the byte parameters are interchanged. The choice with primitive variables is used, since it keeps the interface to the run-time environment most clean.

Instead of using five separate tables for all time signals, the time queue signals could be placed into the job table. This would make only one table for all time signals necessary. This might increase the performance slightly, but probably almost not noticeable. The design would however be much more complicated and harder to understand which makes this a bad choice.

4.8 Signal properties
The only signal property that needs to be considered is buffer/direct. That is because all buffered signals will be treated by the run-time environment and all direct signals will be treated by the compiler. All combinations can be implemented in any of the below mentioned ways.

4.8.1 Buffered signals
Both the CP and the RPs can send buffered signals. When the CP sends buffered signals with a higher priority group than the currently executing job, the currently executing job should immediately be interrupted. This also holds for RP signals in the APZ. But in APZ VM the same requirements are not set on RP signals, which do not need to interrupt the jobs immediately ([6]). Exception to this is if a direct signal is executing, since they can not be interrupted.

Since RP signals do not have these requirements, things can be simplified a lot. When queuing a RP signal all that needs to be done is to put it in the queue and increase the corresponding interrupt counter. If the signal had to be executed immediately, the main-thread would somehow need to be interrupted. The only way to do this in Java is to use the deprecated suspend()-method, which would create the possibility of deadlocks.

The above mentioned issues make a demand of using two different methods for queuing buffered signals. One of them is used for queuing buffered signals sent from the CP and the other one is used by RPs to queue buffered signals.

The internal queue method, queueInternal(), first queues the signal into the correct job buffer and then immediately executes it if it has higher priority than the executing signal.
The external queue method, `queueExternal()`, just queues the signal into the correct job buffer.

If an RP uses the incorrect queue method it would make the job execute in the RP thread. This would cause concurrent execution of jobs, which is not allowed in Plex.

### 4.8.2 Direct signals

The task of sending direct signals is delegated to the compiler. There is no reason to let the direct signals go through the run-time environment. It will be faster to send them to their target block immediately.

The compiler must generate code for sending signals. The easiest way to do this is probably to create the desired signal object and call its send()-method. This would be easy but unnecessary costly since an object needs to be created each time a signal is sent.

Despite what is said in Section 4.6 about using object pools, they might be an optimization in the case of direct signals. There only needs to be one object stored in every of these pools. But on the other hand, the JVM might do a better optimization without using pools. It will at least give less live objects for the garbage collector to bother about. A live object is an object that is referenced by a reference in the program, either direct or indirect through other references.

A better approach for minimizing object construction is to use method-calls. Every block will have to include a method with parameters for each direct signal it is able to receive. To send a signal, the corresponding method is called.

All direct signals are left to the compiler and will not be considered further during the thesis.

### 4.9 Threads

At least two threads must be used in the system.

- At least one thread is used for the processor to execute jobs.
- Another thread is used to simulate RP-CP signals. This thread is not really a part of the run-time environment, but is needed for testing and would be used if RPs are included.
This additional thread is also good to use.

- One thread that simulates the external clock interrupts.

Two different approaches for realizing the execution of jobs were considered. The first used one thread for each of the three priority group levels of jobs while the other used only one thread that executed jobs of all the different priority levels. The reason for trying the second approach was that the first was discovered to be impossible. Below follows a short description of each of the approaches.

### 4.9.1 External interrupt

External clock interrupts are repeatedly sent to the job buffer. The interrupts are called CIS interrupts and are sent at 10 ms intervals.

To simulate these interrupts the Java classes java.util.Timer and java.util.TimerTask ([15]) are used. Time synchronization is done with the scheduleAtFixedRate() method. With this method all CIS interrupts are sent. If any of the interrupts are delayed, it is compensated for by shortening the following interrupts to catch up. This is the behaviour that is expected by the system.

A thread is not really necessary to simulate the clock interrupts. It is possible to poll the system clock and check when 10 ms has passed. But this means that the CPU in the target system would be busy polling the system clock all the time and thereby limiting concurrent execution of other systems. The informal EB tests (see Chapter 5) showed no improvement trying this (after optimizations mentioned later in the chapter).

### 4.9.2 Job execution with three threads

When a CP-CP or RP-CP signal is sent to the APZ, it checks if the signal belongs to a higher priority group than the currently executing job. If that is the case, the currently executing job is immediately interrupted, and the incoming job is started.

The only way to implement this in Java is to use the suspend()- and resume()-methods of threads. Each job priority group also needs to execute in separate threads. This will make it possible to immediately interrupt a job when a new signal arrives. This is the approach that must be used if the APZ should be truly emulated.
As stated above, one thread is needed for each priority group level. This means that three threads will be used for executing jobs. Also a thread for controlling the jobs is needed. This thread will wait for interrupts and eventually suspend the currently executing job, after which the incoming job can be started.

Each of the job executing threads is responsible for executing all jobs in its associated job priority group level. Only one of these threads is allowed to execute at a time, the other threads kindly have to wait for their turn. Higher priority threads can interrupt lower priority threads. For interrupting threads the Java statements suspend() and resume() are used.

It is very good that this approach can interrupt threads without delay and thereby be able to truly emulate the job handling in the APZ. This makes the chances better that old applications will work as they should.

However, there is a big problem. The suspend- and resume-methods are deprecated. The reason to why they are deprecated is that they are extremely deadlock-prone. As soon as synchronization is used, it might result in a deadlock. More about this can be read in [12]. This means that no synchronization can be used. Deadlocks can not be accepted.

Lea [21] states that no synchronization is needed for accesses and assignments to all built-in scalar types except long and double. This makes it look like the approach really could be used by avoiding using synchronizations. It actually can be used, but not in a deadlock-safe manner.

Reference [7] (Praxis 50) and other newer literature state a completely different thing. It states that shared variables always must be accessed synchronized or declared volatile. The JVM is guaranteed to treat reads and writes of data of 32 bits or less as atomic. However, this can not be used in place of synchronization or volatile declared variables. The reason is that Java allows threads to keep private working copies of shared variables.

The volatile modifier might still be used in place of synchronization. The modifier guarantees that any thread that reads a field will see the most recently written value. However, the extent of its applicability is not known until the current work on the JVM memory model is finished ([10] Item 48), which it still not is in J2SE 1.4. Therefore it can not be used in the run-time environment.
This means that this approach not is possible. Synchronization must be used. This in turn means that suspend() and resume() may cause deadlocks. This is a real-time system and therefore deadlocks can not be accepted.

It may seem strange that different literature states different things. But that can be a cause of changes to the Java thread handling between different Java language versions.

There is another approach that does not use suspend() and resume(). The thread priorities built into the JVM could be used and the JVM scheduler could take care of the job scheduling. This seems to work, since the JVM’s thread scheduler acts very similar to the expected behaviour of Plex’s scheduler. They both are preemptive and choose to run the thread with the highest priority. In the case of JVM, this is called fixed priority scheduling. Preemptive means that if a thread with higher priority than the currently executing thread becomes available, the currently executing thread is interrupted and the higher priority thread starts executing instead.

But, according to [14] (the Threads lesson), the JVM thread scheduler sometimes chooses to run threads that not have the highest priority. This is made to avoid starvation. Starvation occurs when threads cannot make any progress since they cannot gain access to a certain resource. This behaviour does not correspond to the expected behaviour for the run-time environment, which makes this approach unusable. If this approach were used, things as emergency calls might have to wait for jobs with much lower priority. This can of course not be accepted.

This gives the conclusion that the APZ job handling currently can not be truly emulated within the JVM.

4.9.3 Job execution with one thread
This approach only uses one thread for executing jobs of all priority levels. The problem with this is that lower priority group jobs somehow need to be interrupted when higher priority group jobs arrive.

Since all jobs are executed in the same thread and suspend() and resume() can not be used (see the approach with three job threads), there is no way to immediately stop the execution of the thread.
Reference [6] states that APZ VM does not immediately interrupt jobs of lower priority. Instead, all jobs except those with the highest priority (since they can not be interrupted), repeatedly checks if any higher priority jobs have arrived. If any higher priority jobs have arrived, the currently executing job is immediately interrupted. The APZ VM is able to execute old APZ applications and therefore this approach is also used in the run-time environment.

The implementation of this approach needs help from the compilation process. During compilation of code that is to be executed in any of the BAL priority group levels, checkpoints repeatedly need to be inserted into the code. A checkpoint should do a method call to the run-time environment to see if any higher priority jobs have arrived. If this is the case, the run-time environment starts to execute all higher priority jobs. The control is not given back until all higher priority jobs are executed.

The requirements say that jobs can turn off other higher priority jobs’ ability to interrupt them. In this case, this is easily implemented by the compiler. The compiler must not generate checkpoints when interrupts are switched off.

When a certain number of instructions have been executed in any of the BAL priority group levels, yield() must be called (see Figure 21). After that, the execution can continue. When a job, which is executing in the BAL levels, sends a buffered signal it should call yield(). If the buffered signal has higher priority, it must immediately start its execution. When yield() has executed all higher priority jobs it returns the control.

![Figure 21. Execution of a C- or D-level job (when interrupts are on)](image)

The compiler will generate the yield() calls when generating the Bytecode. These method calls must be placed in all C- and D-level jobs. It is up to the compiler to decide how often these calls are generated.
Since method calls are expensive, the APZ VM not always calls the yield() method although code is generated for it. A check is first performed and only if a certain time period has passed, the yield() method is called. Such code could be generated by the compiler even though Java is used. But that will only do any good if the currentTimeMillis() method in java.lang.System ([15]) is inlined. Otherwise an expensive method call is done anyway. But since the currentTimeMillis() is a static method it probably will be inlined by the Java compiler or the JIT compiler (praxis 36 reference [7]).

Figure 22 shows how the main-loop checks for interrupts. It also shows which tasks must be synchronized.
Figures 23 and 24 show how the yield() method works. This method is called by jobs executing on C and D level respectively. The figures also show which tasks must be synchronized.
FIGURE 23. Yield-method for C level jobs (executed by Main thread). Text in *italics* must be synchronized.
4.10 Subroutines

There is a problem that only can occur in subroutines. A subroutine is able to be interrupted if it executes on C- or D-level or if it is called by a job that is the result of a direct signal. The subroutine therefore needs to know if it should make yield() calls or not.

This is solved by calling the subroutine with an extra boolean variable. This variable specifies if yield() calls should be made or not.
### 4.11 Synchronization

As stated earlier in the report, all access to shared data has to be synchronized. There are only a few objects that are used by many threads. These are the job buffers (JBA, JBB, JBC, and JBD), their corresponding interrupt attributes (Interrupt JBA, JBB, JBC, and JBD), and the CIS interrupt attribute.

There are four job queues and a job table. The job queues are shared and access to the queue entries need to be synchronized. The job table entries are only used by the job thread and thereby do not need to be synchronized. Here follows which thread uses which queues:

- **Job thread**: JBA, JBB, JBC, and JBD.
- **RP thread**: JBA, JBB, JBC, and JBD.

Five different interrupts can occur. Here follows which thread uses which interrupt attributes.

- **Job thread**: interrupt JBA, JBB, JBC, JBD, and CIS.
- **CIS thread**: interrupt CIS.
- **RP thread**: interrupts JBA, JBB, JBC, and JBD.

When the system is run together with the Optimizeit or PerfAnal profilers (explained in reference [25] and [22] respectively) a tendency for bottlenecks is shown during the locking of objects. The usage of threads in the system is big and therefore the design of these parts is very important. A good design will have a performance effect on the whole system.

Most optimization ideas were based on minimizing the number of synchronizations. According to praxis 34 in [7] this is important for performance. Synchronization is very often used in this code and makes up a part that is suitable for optimization.

A number of design issues are presented below. Some are more important than others.

All optimizations together give a decent performance improvement. The single improvements are not so big. These are the only optimizations the author could think of that would not destroy the design. Optimizations that destroyed the design were not considered. No optimization was found that was worth destroying the
design to implement. It would require a very good optimization if it should be worth destroying the design since that would give many bad consequences for the whole system.

Combinations of the different optimizations were unofficially tested by the EB test (see Chapter 5) extended with additional load from the RP thread. Most of the implemented optimizations gave a few percent better performance and the total increase in performance was about 10-20%. The major reasons for making the choices were not the results of these tests, reasoning was the reason for making the choices. The EB test is reliable, but too specialized to trust when optimizing for running large general applications. The run-time environment is aimed to run large general applications, not just the EB test. It might be possible to optimize more according to testing with the EB test, but since the EB test does not test the whole run-time environment this could have bad consequences for the run-time environment. The only reason that the unofficial tests were actually performed was to give an indication that the implemented optimizations did not give worse performance than before.

A bigger and more complete test must be constructed if formal tests are about to be used for making choices during code optimization.

4.11.1 Placement of interrupt indicators
The interrupts need to be saved either in one object each or in the same object. Good candidates for this are to either save the queue interrupts in the associated queue or save them all in the job handler. The CIS interrupts are preferably saved in the job handler.

After looking at both advantages and disadvantages with the first and second approach, the first approach was chosen. The first approach is faster at scanning for incoming interrupts while the second approach is faster when queuing signals. An important addition is that wait() and notify() only can be used by the first approach.

Save all interrupts indicators in job handler
When all interrupt indicators are placed in the job handler, all interrupts can be checked with one synchronization. This makes it possible to use the wait() and notify() methods, which takes away load from the processor.
The job handler will be locked often, since every interrupt that is added or removed will lock it. Only one interrupt can be added or deleted at a time. The CIS interrupt occurs very often and makes locking occur even more often.

Since the jobhandler is locked more often than the second approach, the queuing of signals takes some more time.

**Save all interrupts in the associated queue**

In difference to the other approach, here interrupt indicators can be added or removed concurrently even if other interrupt levels are locked.

Interrupts can not be checked with one synchronization since queuing can be done concurrently. During the checks, a new signal could be queued in an already checked queue. This will make the result incorrect. This makes it impossible to use wait() and notify().

This approach makes the scanning of interrupts slower since more synchronizations with the queues have to be done. This will slow down C- and D-level jobs when they call yield(). This approach wastes many processor cycles, since wait() and notify() are not used. This could slow down the whole system and make it hard for example for the garbage collector to perform its task. When notify() and wait() are used, the JVM do not use many resources. This makes it easier for the system that runs the JVM to perform tasks, and thus makes the chances smaller that these tasks interfere with the run-time environment (since they are hopefully made during the wait()).

The only things that can be done concurrently while the job handler is waiting for incoming interrupts are that the RP thread can queue signals and that the CIS thread can send interrupts to the job table. These tasks will certainly be slowed down if polling loops are used instead of wait() and notify(). This will probably have some impact since the CIS thread sends interrupts every 10 milliseconds.

**4.11.2 Interrupt flag or counter**

Another question to consider is if all job buffer interrupts have to be recorded or if only those that occur when a queue is empty have to be recorded. If all interrupts are recorded a counter is used, if only interrupts are recorded when a queue is empty a flag is used.

Using flags for the interrupts is better, since the operations for adding and removing interrupts is synchronized and thereby expensive.
**Interrupt counters**

When using this approach, no checking of the queue needs to be done before adding elements. Just add every time an interrupt arrives.

The addition and removal of interrupts are expensive synchronized operations.

**Interrupt flags**

Every time a signal is queued, the queue needs to be checked if it is empty or not. Interrupts are only added if the queue is empty. When a signal is removed from a queue, the queue also needs to be checked. If it becomes empty, the interrupt flag must be cleared.

4.11.3 Placement of CIS counter

There must be a counter for the CIS interrupts, since the jobtable does not contain anything that shows how many interrupts have occurred. This can be compared to the job buffers which contain a certain number of signals.

This counter can either be placed in the job handler or in the job table. The timer that generates the CIS interrupts is preferably connected to the same object as the counter.

To avoid unnecessary locks, the counter is placed in the job table.

**Counter in job table**

If the counter is placed in the jobtable, both the method for incrementing the counter (when CIS occurs) and scanning the job table need to be synchronized. But this also means that the job handler does not need to be locked at every CIS interrupt. The CIS interrupts occur very often, so the choice of placing the counter in the jobtable is better in most cases, this was implemented in the system.

**Counter in job handler**

In this approach the jobhandler must be locked every time a CIS occurs. This is bad for the performance.
4.11.4 Series of job table scans
If more than one CIS interrupt occurs between two scans of the jobtable, all these
scans will occur in a series without interrupt. This is because the job table has the
highest priority. It feels very unnecessary to actually synchronize the interrupts
between these scans, so an optimization is to scan the whole series without
synchronization of the interrupts. This optimization was implemented. The impact
of this optimization will be the bigger the longer the jobs are. That is because more
CIS interrupts will occur during a long job.

4.11.5 Job table flag
Another thing to consider is if the job table or the job handler should be aware of
whether the table needs to be scanned or not. If the job handler is aware of it, by
placing a flag in the jobhandler, the table only needs to be called when an interrupt
has occurred. In the other case the table is called every time, and it decides by itself
if it is going to be scanned or not (this makes the use of wait() and notify() impossible). When placed in the job table there will be method calls every time,
even when there is no need to scan the table. Even though the job table chooses to
not scan the table, method calls take time. When instead placing a flag in the job
handler there will also be method calls for setting and clearing the flag, but this will
not be as often as in the other case. This choice is used in the system, not mainly
because of the speed but mostly for releasing the processor with the wait() and
notify() statements. This will probably gain speed when larger systems use the
run-time environment and for example garbage collection needs to be done.

4.11.6 Removal of CIS thread
The CIS-thread could actually be removed. The system can check if a CIS-interrupt
has occurred by checking the system time. If there is a difference of at least 10 ms
from the latest interrupt, a new CIS-interrupt must have occurred.

If the CIS-thread should be removed, wait() and notify() can not be used, the system
would instead have to repeatedly poll the system time. This would limit the uses of
the system since it would constantly consume very much system resources.

The informal tests showed no indications of gain in performance when the
CIS-thread was removed, sometimes the performance even got worse than with the
CIS-thread.
There seem to be mainly disadvantages of removing the CIS-thread and it is therefore kept in the system.

4.11.7 Optimization overview
Figure 25 shows how the loops are constructed after the above mentioned optimizations have been implemented. The loops that are changed are those shown in Figures 22, 23, and 24.

4.11.8 Deadlock analysis
It is very important that no deadlocks can occur in the system. A deadlock would be a disaster, especially since this is a real-time system. This section analyses the threads, locks, and shared objects to confirm that no deadlocks can occur.

These are all the methods that use synchronization in the system. It is also specified which threads that use them.

- JobTable.cis() - CIS thread.
- JobTable.scan() - Job thread.
- JobBuffer.front() - Job thread.
- JobBuffer.queue() - Job thread, RP thread.
- JobHandler.syncInterrupts() - Job thread.
- JobHandler.setInterrupt() - Job thread, RP thread, CIS thread.
- JobHandler.clearInterrupt() - Job thread.

The following are shared objects. The threads that use them are also specified.

- Interrupt flags A/B/C/D - Job thread, RP thread.
- Interrupt flag Table - Job thread, CIS thread.
- Queue A/B/C/D - Job thread, RP thread.
- CIS counter (in job table) - Job thread, CIS thread.

The three threads lock objects in the order given by the following figures. An oval in the figures represent an object that is locked. If the oval is dashed, the lock is not always created. Locks are always released in the opposite order of how they were acquired.

**CIS thread**

The CIS thread does not do much. It only generates a clock interrupt every 10 milliseconds. This interrupt is sent to the job table, by calling the cis()-method. This method is synchronized because it increments the CIS counter. If the cis() method finds that there currently are no not-performed CIS interrupts, it sets the interrupt flag for the job table in the job handler. This method is also synchronized since it is used by all threads. This is shown in Figure 26.

**RP thread**

The RP thread is able to queue signals in the job buffers. Each buffer needs to be synchronized and each of them has a corresponding flag in the job handler that also needs to be synchronized. When a signal is queued, the corresponding job buffer is locked (the other buffers are not locked). If the buffer was empty a flag in the job handler is set, during which the job handler is locked. This is illustrated in Figure 27.
Job thread

When executing the main-loop, all interrupts are checked at the same time during an iteration. This is done to prevent it from happening at every interrupt level, and to be able to use wait() and notify(). This is shown in Figure 28.

When a signal is about to be executed, it is first fetched from the queue and then executed. This needs to be done in two steps. The signal must be executed from the job handler, not from the queue. If the signal is executed from the queue, the queue would remain locked during the execution of the signal. If the signal then would try to place another signal in the same queue, a deadlock would occur. The locks placed during the fetch of the signal are shown in Figure 29. The signal is removed from the queue during the fetch. If the queue becomes empty, the interrupt flag in the job handler is cleared.

When a signal is queued, it is put into the specified job buffer. The job buffer is thereby locked. If the job buffer was empty, the flag for the specified job buffer is set in the job handler. The job handler is also locked when this happens. See the illustration in Figure 30.
During the main loop, it is checked if any CIS interrupts have occurred since the last scan of the job table. If this is the case, the job table is scanned. Since the job table has the highest priority, it can not be interrupted. Because of this, all scans of the job table that are needed are done immediately. After that the CIS interrupt flag is cleared in the job handler.

The above mentioned cases of locks can not get the system into a deadlock. That can be seen from the fact that the locks always are acquired in the same order. Either a jobbuffer or the jobtable, then the jobhandler. The order is never opposite, which prevents a deadlock from occurring (according to reference [32]).

### 4.12 Other design issues

This section describes some issues in the design that do not fit in to be described elsewhere.

#### 4.12.1 Date conversion

For converting between dates, the millisecond-methods in java.util.Calendar ([15]) were used. This was to avoid the otherwise big overhead of using Date-objects. The millisecond-methods have public access in J2SE 1.4, but in J2SE 1.3.1 they have protected access. This means that the methods can not be used in J2SE 1.3.1. This is very bad since the target system for the run-time environment uses J2SE 1.3.1. But Sun changed the methods access modifiers to public in the J2SE 1.4 release. The method implementation was not changed. For further information see reference [26].

Since the methods have protected access in J2SE 1.3.1, a workaround could be written. This was done by making a new class (called GregorianCalendarExtended) that extended java.util.GregorianCalendar ([15]) and which overrode the methods and made them public. It should actually be more appropriate to make them have package access, but this can not be done since the methods have public access in J2SE 1.4 (which would then make the system not forward compatible). But this is not very important since the class has package access. This new class was used by the absolute time queue instead of java.util.GregorianCalendar. This workaround
could not safely have been used if Sun also had changed the method implementation.

4.12.2 Robust
The run-time environment is designed to be robust. All methods that have public access throw exceptions if the preconditions are violated. Possible exceptions that are thrown are java.lang.IllegalArgumentException, java.lang.IndexOutOfBoundsException and java.lang.NullPointerException. Methods with package or private access do not throw any exceptions. But the preconditions for those methods were checked during development by using assertions (specified in reference [27]).

4.13 Usage
Here follows a short description of how the run-time environment is supposed to be used. All methods and classes that are supposed to be used by RPs and other Plex blocks are described. All access to the job handling in the run-time environment goes through the JobHandler class. In order to avoid sending the reference to the instance of this class to everyone that needs it, the class is implemented as a Singleton. The instance is therefore globally accessible.

Both the RPs and the Plex blocks need access to the interfaces Signal and Block, otherwise they can not create signals and blocks.

4.13.1 Regional processors
The only way in which the RPs can interact with the run-time environment is to send buffered signals.

Job buffers
- queueExternal() inserts a RP signal into the specified job buffer.

4.13.2 Central processor
Code in the CP has much more requirements than code in the RPs. It should be able to use the job table, the job buffers, and the time queues.

When jobs of priority BAL 1 or BAL 2 are executing, they should repeatedly call the yield() method. This method checks that no jobs with higher priority have
arrived. If this is the case, all higher priority jobs are executed before control is returned.

**Job table**
- `jobTableInsert()` inserts a signal into the job table.
- `jobTableRemove()` removes a signal from the job table. This can only be done on sent signals.
- `jobTableSetInterval()` changes the interval of a signal that is placed in the job table. This can only be done on sent signals.

**Job buffers**
- `queueInternal()` inserts a signal into the specified job buffer. It interrupts the current job if the queued job belongs to a group with higher priority. This method should only be used by CP-CP signals.
- `queueExternal()` inserts a signal into the specified job buffer. It does never interrupt the current job. This method should only be used by RP-CP signals.

**Time queues**
- `queueRelative()` queues a signal into the specified relative time queue.
- `queueAbsolute()` queues a signal into the absolute time queue.

**Other methods**
- `yield()` is invoked by C- and D-level jobs. It interrupts these jobs if any jobs that belong to groups with higher priority are available. The method returns when all jobs that belong to groups with higher priority have finished executing.

### 4.14 Class descriptions

This section gives a short description of the classes that are included in the run-time environment and the interfaces that are used by applications.

Figure 32 shows the *class diagram* for the run-time environment. The parts drawn with dashed lines are not really part of the run-time environment but must be used by applications.
Boxes written with plain letters represent *classes* and boxes written with italics represent *interfaces*.

![Class diagram]

**Main**

The Main class starts the run-time environment. It creates the job handler and sends initial signals.

**JobHandler**

The JobHandler class handles the execution of jobs. It creates the different queues and schedules the jobs. It also represents the visual interface for usage of the run-time environment.

**Block**

The Block interface represents the behaviour that must be implemented by every Plex block.
Signal
The signal interface represents the behaviour that must be implemented by every Plex signal.

JobBuffer
The JobBuffer class implements a Plex job buffer.

JobTable
The JobTable class implements the Plex job table.

TimeQueueAbsolute
The TimeQueueAbsolute class implements the Plex absolute time queue.

TimeQueueRelative
The TimeQueueRelative class implements a Plex relative time queue.

ClockInterrupt
The ClockInterrupt class simulates the CIS (Clock Interrupt Signal). It sends an interrupt every 10 ms.

GregorianCalendarExtended
The GregorianCalendarExtended class implements the java.util.GregorianCalendar, and makes sure that the millisecond methods are not protected.

4.15 Javadoc
The class description in Section 4.14 is not detailed. To provide a more detailed view of the system Javadoc documentation was generated.

Javadoc documentation for all public and package methods and attributes in the run-time environment is included in Appendix A. There also exists Javadoc documentation for all private methods and attributes, but this is not included since it is not considered relevant for the reader. The Javadoc tool and the MIF Doclet ([23]) were used to generate documentation in MIF (Maker Interchange Format).
Javadoc for all public and package methods and attributes gives enough information of the implementation, therefore the source code is not included in the thesis.
5 Test

This chapter describes how the system was tested. The testing was done by using a benchmark test called EB (Emulator Benchmark).

An already existing test was used since the construction of a realistic test is a complicated issue. This thesis did not include time for such a task.

The EB test was chosen because of its relatively small size and that it tested the run-time environment in a quite good way.

All tests were performed on Tru64 UNIX running on a multiprocessor architecture of Compaq Alpha processors. The platform was extended with the APZ VM and the Compaq Fast VM. This is further explained in Section 5.7.

Because of confidentiality reasons no APZ VM time performance measurements are presented in the chapter. Diagrams are drawn to show the relative speed of the two systems. But all measurement values are placed in tables in Appendix B, which is confidential.

5.1 EB

The EB test is originally constructed for measuring performance of APZ emulators and comparing them to a real APZ. The test is designed to be realistic by resembling real-world traffic handling situations ([11]), which makes it good to use for comparing the two different systems in a fair way. The EB test takes a number of parameters as input. By varying these parameters, the EB test can be used to test different aspects of the system.

The author used the test for measuring processing performance of his system and then comparing it to a real system.

The author used the EB test version 2. A description for the EB test version 2 is not available, but a description for the similar EB test version 3 can be found in reference [11].
5.2 Construction

The EB test is written fully in Plex-C. Since the constructed system is made for running Bytecode, the test code had to be compiled to Bytecode. Since there does currently not exist a Plex-C to Bytecode compiler there were two options. Either compile the Plex-C code by hand to Bytecode or translate the Plex-C code by hand to Java and compile it with a Java compiler.

The obvious choice is the last one. The handcompilation of Plex-C code to Bytecode is beyond the scope of this thesis and it would be far too time-consuming. The choice of translating the Plex-C code to Java might make the resulting Bytecode a little less effective, this is considered by the author to only have insignificant influence on the measurements since the JIT optimizes the code during execution.

The translation of the EB code to Java was successful. Some problems occurred since Plex-C allows the use of the goto construct. This could however in all cases either be eliminated or translated into loops which make use of the break and continue statements (for more information see 7.6 - 7.10 in reference [16]).

The blocks are connected into a chain. Every block in the chain is modelled as a class. In Java it would have felt more natural (and easier) to model each block as an instance of a class. But the compiler will model the chain as classes, which is the reason for modelling as classes. The only difference between different intermediate blocks is the class name and the references to the previous and next block.

Precautions were taken, to do the conversion as similar as possible to how a compiler would have done it. No optimization was made, just a standard translation according to Chapter 4. The translation was done very carefully, since an undiscovered error could imply very misleading results from the test.

The Compaq Fast VM compiler makes no Bytecode optimization to the EB test code during Java to Bytecode compilation. This has been verified through decompiling of the .class-files, using JAD 1.5.8e [17], and then comparing the results with the .java-files (the source code).

The target processor was assumed to take care of R variables as 32-bits variables. This makes R variables become variables of type long.
5.3 Description

The EB test uses totally 33 blocks and 10 signals.

The blocks are connected into a chain. Every block only has references to its closest neighbours. This is shown in Figure 33. There are two kinds of blocks:

- **EBFIRST.** This is the block that controls the whole test. It sends signals to the EBNXT blocks, according to the input parameters.

- **EBNXT.** All EBNXT blocks are exactly equal, except their neighbour references. A parameter specifies how many intermediate blocks will be included in the test. The last block that is included is called the last block, and the blocks before that block are called intermediate blocks. The EBFIRST block sends signals that propagate through the intermediate blocks, until they reach the last block. The last block sends signals in the other direction.

![Figure 33. EB test blocks](image)

Every block has a table for storing phone calls. All EBNXT blocks also have a table for storing destinations for the phone calls. The destination table is only needed in the last block, but is also included in the intermediate blocks to increase the job time.

Every used entry in the table for storing calls holds a pointer to the same call in the table in the next block and the previous block (except the last block which does not have a next block). The entry in the last block also has a pointer to the destination in the destination table.

The destination table contains a series of phone numbers. They are ordered in increasing order. This table is only needed for the last block, but it is included in every block anyway. The reason is that every block does a number lookup in the table (even though it only needs to be done by the last block) to increase the total job time.
All signals that are used in the test are in some way propagated through the chain of blocks.

- **STME.** This signal is sent to EBFIRST to start the test. All test parameters are included in the signal.

- **STMER.** This signal is sent back to the system to finish the test. All statistics for the test are included in the signal.

- **SETUP.** This signal sets up a phone call through the chain of blocks. The signal is sent from EBFIRST and stopped in the last block. The signal contains a phone number. At every block, an entry for the phone call is allocated in the call table. This entry is set to point to the previous block’s entry of the call. At every EBNXT block a lookup for the phone number is made in the destination table (the lookup uses a binary search algorithm). This is only necessary to do in the last block, but is also done in all intermediate blocks to increase job time. If the lookup is performed successfully in the last block and the destination is not busy, a SETUPR signal is sent back through the blocks. The call is marked as busy in the destination table. There can however occur errors. If any error occurs, a RELCOMP signal is sent back through the blocks, specifying which error occurred. At every block, the entry for the phone call is deallocated. Possible errors in the first intermediate block are: blocked (there was no free entry in the call table). Possible errors in the last block are: not found (the destination phone number for the call can not be found in the destination table), busy (the destination call entry in the last block has marked the call as busy).

- **SETUPR.** This signal is sent from the last block, and then propagates through the blocks until it reaches EBFIRST. The signal completes the setup of the call by also let the allocated call entry in each block point to the next block’s entry of the call. When the signal reaches the EBFIRST block it is checked if any DATAFWD signals have to be sent. If not, a RELEASE signal is sent through the blocks. Otherwise a DATAFWD signal is sent through the blocks.

- **RELCOMP.** This signal is sent to the EBFIRST block, specifying the status for how an operation failed or succeeded. At every block, the entry for the call is deallocated. When the signal reaches EBFIRST, the return code is examined and the corresponding counter is incremented. If all calls are finished the status for the test is sent back to the system with a STMER signal, containing all statistics for the test.

- **RELEASE.** The purpose of the signal is to release the call. This signal is sent from EBFIRST to the last block. When it reaches the last block, the destination for the call is marked as not busy. The call is then deallocated from the call table.
A RELCOMP signal is sent back through the blocks, telling it was a successful call.

- **DATAFWD.** This signal is sent from EBFIRST to the last block. At every intermediate block, a certain number (specified as test parameter \textit{inddata}) of variables from the call in the call table are accessed. When the signal reaches the last block, a DATABWD signal is sent back through the blocks.

- **DATABWD.** This signal is sent from last block to EBFIRST. When it reaches EBFIRST, it is checked if all specified number of DATAFWD signals have been sent. If not, another DATAFWD signal is sent through the blocks. If all DATAFWD signals have been sent, a RELEASE signal is sent through the blocks.

- **CONTINUEB.** This signal is inserted into the job table to add additional execution time to other jobs waiting for the processor. This signal is inserted a certain number (specified as a test parameter) of times into job table B between new calls. When running test on C level, these signals are inserted by the CONTINUEC signal. This is called \textit{phase division} in Plex.

- **CONTINUEC.** This signal is inserted into the job table to add additional execution time to other jobs waiting for the processor. This signal is inserted a certain number (specified as a test parameter) of times into job table C between new calls. It is only used when the test is run at C level. This is called \textit{phase division} in Plex.

A successful call includes sending a SETUP signal to the last block. This block returns a SETUPR signal. This is followed by a series of DATAFWD - DATABWD pairs of signals. After that, the call is released by sending a RELEASE signal to the last block. This is replied by the last block sending a RELCOMP signal to EBFIRST. The whole process is illustrated in Figure 34.
A call sometimes fails. This happens after the SETUP signal is sent. The fail either occurs in EBNXT00 or in the last block. When an error occurs a RELCOMP signal is immediately returned to EBFIRST. Figure 35 shows a blocked call and Figure 36 shows a busy or not found call.
Figure 37 shows one of the calls that can possibly be setup in the test. Many calls are normally set up at the same time. The table up to the left represents the call table. The table down to the right represents the destination table. The destination table only has a real use in the last block, but is searched through in every block to consume some CP cycles.

The statistics that are received from the test are the following:

- **Successful.** The number of successful calls.
- **Busy.** The number of calls to busy destinations.
- **Blocked.** The number of blocked calls. Blocked calls occur when there are no more free entries in the table that holds the calls.
- **Notfound.** The number of calls to unknown destinations.

The above statistics are not so interesting *during* the testing of this system. They were mainly used to see that the test was performed correctly. The time measurement is the interesting thing. The time is not measured by the test. It is instead measured by the system. The resulting time is the system time difference of a point exactly before sending the STME signal to the test and a point exactly after the STMER signal is received from the test.

The statistics is interesting for controlling if the test is correctly implemented in Java. The values can be used to calculate the number of signals that should have been sent and compare to the number of signals that actually was sent in the Java test (by temporary putting a counter into the run-time environment). The statistic values can also be compared between the two tests to see that they have behaved the same way.

Before each call a number of CONTINUEC and/or CONTINUEB signals are sent. This is to make other signals get a chance to execute (phase division). Test parameters specify this.

The total number of signals, N, sent during a test can be calculated. This can be used to understand how the signals are sent and later on for testing of correctness of the system.

- \( N(\text{SETUP}) = (\text{blocks} + 1) \times (\text{successful} + \text{busy} + \text{notfound}) + \text{blocked} \)
- \( N(\text{SETUPR}) = (\text{blocks} + 1) \times \text{successful} \)
- \( N(\text{DATAFWD}) = (\text{blocks} + 1) \times \text{successful} \times \text{sigs} \)
- \( N(\text{DATABWD}) = (\text{blocks} + 1) \times \text{successful} \times \text{sigs} \)
- \( N(\text{RELEASE}) = (\text{blocks} + 1) \times \text{successful} \)
- \( N(\text{RELCOMP}) = (\text{blocks} + 1) \times (\text{successful} + \text{busy} + \text{notfound}) + \text{blocked} \)
- \( N(\text{CONTINUEB}) = \text{conts} \times \text{calls} \)
- \( N(\text{CONTINUEC}) = 0 \) if lev=JBB. \( \text{conts} \times \text{calls} \) if lev=JBC.
- \( N(\text{STME}) = 1 \)
• \( N(\text{STMER}) = 1 \)

The total number of sent signals during a test is:

\[
N(\text{total}) = 2*(\text{blocked}+1+(\text{blocks}+1)*((2+\text{sigs})*\text{success}+\text{busy}+\text{notfound})) + \text{conts}\*\text{calls}
\]

If the lev-parameter is equal to JBC, an additional \( \text{conts}\*\text{calls} \) will be added to the sum.

All the variables in the formula are either in- or out-parameters of the test.

### 5.4 Parameters

There are 8 different parameters that can be used for the test. They are described in Table 4.

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
<th>range</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>calls</td>
<td>Number of calls to execute.</td>
<td>1 - 64K</td>
<td>3</td>
</tr>
<tr>
<td>blocks</td>
<td>Number of intermediate EBNXT blocks.</td>
<td>0 - 31</td>
<td>0</td>
</tr>
<tr>
<td>sigs</td>
<td>Number of DATAFWD signals that will be sent for each call.</td>
<td>0 - 64K</td>
<td>2</td>
</tr>
<tr>
<td>indata</td>
<td>Number of variables that will be accessed in each intermediate block, in the entry for each call.</td>
<td>0 - 6</td>
<td>2</td>
</tr>
<tr>
<td>maxtelno</td>
<td>Number of destinations. This is the maximum index that is used in the binary search.</td>
<td>1 - 2000</td>
<td>2000</td>
</tr>
<tr>
<td>conts</td>
<td>Number of CONTINUE signals sent before each new call.</td>
<td>1 - 64K</td>
<td>1</td>
</tr>
<tr>
<td>lev</td>
<td>Priority level for generating CONTINUE signals (CONTINUEB or CONTINUEC). When JBB is sent, only CONTINUEB is sent. When JBC is chosen, first CONTINUEC is sent, then CONTINUEB.</td>
<td>JBB, JBC</td>
<td>JBB</td>
</tr>
<tr>
<td>seed</td>
<td>Seed for random number generator.</td>
<td>0 - 64K</td>
<td>1</td>
</tr>
</tbody>
</table>

### 5.5 Test cases

The only test case that was documented for EB2 was the default values found in the source code. These values were not usable. When the test was run with these values it completed so fast that the values most often were not even measurable.

Reference [11] provides some suitable test cases, which were used for testing the system. Since these values are specified for EB3, they had to be adapted to EB2. The values of some parameters in EB2 were not specified and some values for
parameters that did not exist in EB2 were specified. According to creators of the EB test, EB3 was only adding functionality to EB2. Therefore the same parameters can be used. The values were adapted by setting the values that were not specified to their default values and the parameters that did not exist were not used.

Test case 8 acts like the EB1 test. This makes it the most usable test case, since the other test cases are extensions to the EB2 test.

All the different test cases are shown in Table 5. The table shows the test cases together with the corresponding parameters.

<table>
<thead>
<tr>
<th>calls</th>
<th>blocks</th>
<th>sigs</th>
<th>inddata</th>
<th>maxtelno</th>
<th>conts</th>
<th>lev</th>
<th>seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>30</td>
<td>3</td>
<td>3</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>30</td>
<td>3</td>
<td>3</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>3</td>
<td>60000</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>4</td>
<td>60000</td>
<td>30</td>
<td>0</td>
<td>3</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>30</td>
<td>3</td>
<td>0</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>6</td>
<td>10000</td>
<td>30</td>
<td>3</td>
<td>6</td>
<td>2000</td>
<td>2</td>
<td>JBB 1</td>
</tr>
<tr>
<td>7</td>
<td>10000</td>
<td>30</td>
<td>3</td>
<td>3</td>
<td>2000</td>
<td>100</td>
<td>JBB 1</td>
</tr>
<tr>
<td>8</td>
<td>60000</td>
<td>30</td>
<td>2</td>
<td>2</td>
<td>2000</td>
<td>1</td>
<td>JBB 1</td>
</tr>
</tbody>
</table>

The virtual machines were not shut down during the running of a series of a test case, which consist of 10 measurements with the same parameters. This was to let the JIT compilers warm up. But between every series of measurements the virtual machines were shut down. This was to make the preconditions equal for the both VMs.

5.6 Limitations

There are some limitations with the performed testing. To know the concrete effect of these limitations further testing must be done. The run-time environment has limited functionality compared to the APZ and the EB test only tests a limited part of the system.

Even though these limitations probably have at least minor effect on the test results, the results are still sufficient for giving a good idea on how fast a system built on the JVM is compared to the APZ.
5.6.1 Limited functionality
The functionality of the run-time environment is limited compared to that of the APZ.

- The run-time environment is designed to be expandable. But since not all of the APZ and Plex could be considered in the thesis, some important aspect might have been missed. This could lead to design problems when later expanding the system. If major design changes have to be done, performance loss might occur.
- When the run-time environment grows it might give problems for the JVM concerning memory handling and run-time optimizations. This would in turn give loss in performance.

5.6.2 Limited test
Since the EB test is designed for measuring performance of APZ emulators it is suitable to use for testing the run-time environment. But the EB test does not test all aspects of the run-time environment. Most notably the yield() method is not tested and no RP-signals are sent. The following things are not tested at all or not thoroughly.

- The test almost always executes on the highest group priority level. Even if the lev-parameter is set to JBC no yield() calls are necessary because of the very short time the test is executing on the lower group priority level. This means that no yield() calls are needed. The yield() method is thereby not tested at all.
- There are no RP-signals. This removes one moment of concurrency. Synchronization is still used and objects are locked, but the risk of waiting is less than usual. However, the clock interrupts are still there and provide concurrency. The use of RP signals will probably not have a major impact on performance, since the Compaq Fast VM uses separate processes during execution of Java-threads. According to informal tests the execution of the signals will not have almost any bad performance (tested by putting a very demanding constant job on the RP-thread), it is only the synchronizations when RP-signals are received that will have a (small) bad impact. All this holds for RP-CP signals. CP-RP signals will not make any noticeable performance impact at all, since they queue in a separate job buffer.
- There are no delayed signals. Potential problems can arise when inserting signals, which the test does not show.

Note that the lack of the above mentioned aspects do not necessarily give the run-time environment an advantage compared to the APZ VM. This is because both
the run-time environment and the APZ VM do not have to test the mentioned aspects. The time performance effects of these limitations are probably approximately the same on both systems.

5.7 Conditions
The performance of both systems was measured on the same platform to increase the reliability in the measurements. The platform was extended with a JVM and an APZ VM. To further increase reliability, as few as possible other applications were running concurrently with the test.

Used JVM: Compaq Fast Virtual Machine (Fast VM) 1.3.1 for Tru64 UNIX (information about this JVM can be found in reference [24]).

Used APZ VM: The used APZ VM includes a JIT compiler. For more exact version information see Appendix B.

Used platform: Tru64 UNIX running on a multiprocessor architecture of Compaq Alpha processors.

5.8 Testing
It is very important that the conversion of the test-code from Plex to Java is correct. Therefore, much additional testing was made to verify that this was the case. The tests verified that:

- All out-parameters were correct. I.e. for equal in-parameters, both the JVM test and the APZ VM test gave equal out-parameters.
- The number of sent signals was counted by the computer in the Java version. That number was then verified against the calculated value.

That both test cases above were verified and always correct gives a very strong indication that the test system works as it should. That the number of sent signals were correct is very important since the test basically consists of sending a large amount of buffered signals.

5.9 Compaq Fast VM
The Compaq Fast VM has a Just-In-Time (JIT) compiler.
Below follow explanations and tests for how choices were made to adjust the Compaq Fast VM.

One important thing to note is that not all JVMs support the same choices as the Compaq Fast VM.

All tests use the EB1 (number 8) test case which is a test case that acts like the first version of the EB test. This is the most usable test case for making these comparisons, since the other test cases are extensions to EB2.

5.9.1 Classic VM versus Fast VM

There are actually two JVMs included in Compaq’s kit.

- The *classic virtual machine* contains Just-In-Time (JIT) compiler technology and has additional debugging support.
- The *fast virtual machine* contains Just-In-Time (JIT) compiler technology but does not have additional debugging support. This virtual machine is the default choice.

Figure 38 and Table 8 show the EB1 (number 8) test case tried on the two different JVMs. They clearly show that the classic VM is not an alternative. The fast VM is therefore used.

![Test case EB1 measured with fast (diamond) and classic (triangle) JVM](image)
5.9.2 Fast32 versus Fast64
The Compaq Fast VM has two different modes for storing pointers into objects. One uses 32-bits pointers and the other uses 64-bits pointers.

The pointer sizes limit the amount of virtual memory available to Java applications. 32-bits pointers should be well enough for the run-time environment. But according to the measurements in Figure 39 and Table 9 the 64-bits pointers give a faster system and are therefore chosen. This is probably due to some internal conversions in the case with 32-bits pointers.

5.9.3 Garbage collection
The performance of garbage collection is mainly measured in **throughput** and **pause time** [9].

The **throughput** measures how much of the total time that is spent on *not* garbage collecting. This must be considered during long time periods. The throughput includes time spent during allocation.

The length of the pauses that occurs during garbage collection are called **pause times**.

The need for different applications is different. Since this is a real-time system, it demands small upper bounds on pause times and also high throughput. Small pause...
times are favoured over high throughput. It is important with high throughput, but too long pause times are not acceptable.

Here follow short explanations of some of the major *indirect garbage collectors*. See reference [18] for more thorough explanations. In indirect garbage collectors the information that is needed to determine if an object can be recycled is not stored with the object. It is instead determined by tracing all reachable objects.

- **Mark-sweep garbage collection.** Traces all live objects and marks them. This is followed by a sequential sweep through the memory during which all unmarked objects are recycled. The recycled objects are added to a free list. No objects are moved during collection.

- **Mark-compact garbage collection.** Traces all live objects and marks them. This is followed by compacting the marked objects into a block, to remove the fragmentation. All objects outside the block are recycled.

- **Copying garbage collection.** Copy all live objects into a free memory area. All the original objects are then recycled.

The Compaq Fast VM supports two different kinds of garbage collectors (reference [24]).

- **Copying collector.** An ordinary copying garbage collector (see above).

- **Compacting collector.** This collector avoids copying data. It instead tries to compact live data in-place. It is a combination of a mark-sweep/mark-compact collector. This is a multi-threaded collector that can scale on multiprocessors.
Figure 40 and Table 10 show that the compacting garbage collector has a higher throughput than the copying collector.
The test values for Table 6 (and Figure 41) were received by using the standard JVM option for doing verbose garbage collections.

In the first test run in Table 6 (and Figure 41) the compacting garbage collector does a *compacting* collection, while in all the other test runs it does a *sweep* collection. That is the reason for the big difference in pause times between the first test run compared to the others.

The *copying garbage collector* has very static pause times. Further measurements show that pause times lie between 0.05 and 0.08 seconds.

The *compacting garbage collector* most of the time has static pause times which lie between 0.15 and 0.17 seconds (shown by further measurements). But when it does *compacting* collection there are big peaks.

The small gains that are given in throughput of using the compacting garbage collector are not worth the increased pause times and sometimes high peaks that occur in the pause times. Therefore the copying collector is chosen. Reference [24] states that the compacting collector can give better performance and lower heap size requirements than the copying collector if the heap mainly contains long-lived data. The *heap* is the memory area that is managed by dynamic allocation, also known as *free store*. The run-time environment mainly contains long-lived data. But the amount of long-lived data is small. That is probably the reason for why the compacting garbage collector does not show smaller pause times than the copying garbage collector. The copying collector would probably not perform equally well for a larger application. In the case with a larger application the compacting collector might give a better result. The multiprocessor did not seem to help the compacting garbage collector that much.

Since the copying collector shows much better results during testing, it is used in the system.

### TABLE 6. Garbage collection times for copying and compacting GC. Test case EB1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>copying</td>
<td>0.08</td>
<td>0.08</td>
<td>0.06</td>
<td>0.05</td>
<td>0.07</td>
<td>0.05</td>
<td>0.05</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>compacting</td>
<td>0.70</td>
<td>0.16</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.16</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
</tbody>
</table>
The fast32 choice can sometimes give slightly smaller pause times. But not so much that it is becoming the preferred choice.

The garbage collector in the classic virtual machine starts with very small pause times. But only after a few seconds they have raised to a constant rate of 98 to 100 milliseconds. Just as stated above, the classic VM is still not an alternative.

The EB test is suitable for testing garbage collection in the aspect that it sends a large number of signals. But it is on the other hand bad since the static part of the test takes up a quite small amount of memory. A very large test may show the need of using the compacting garbage collector. This could in turn cause trouble when compacting collecting occurs.

The compacting garbage collector seems to do compacting collection only the first time. This is after all blocks and other initializations have been performed. Afterwards the amount of live objects stops to grow. It only grows temporarily and then goes back to the same amount again. A bigger test is probably needed to test this thoroughly. But if only sweep collections need to be done after the first collection, this might be a better alternative than the copying garbage collector. The amount of live objects does not increase by time, but might be fragmented and thereby making the need for a compacting collection. The amount of live objects temporarily increases but always goes down again without any compacting collections.

5.9.4 Heap management
The virtual machine also has options for setting the initial and maximum amount of heap memory. Tweaking these parameters might give an improvement in speed and maybe shorter pause times for the garbage collector. But that would be specific for this test (not the run-time environment) and these settings are therefore left intact. If unspecified, the Compaq Fast VM sets these values according to the maximum amount of memory available ([24]). The default initial heap size is set to 10% of the maximum amount of memory available. The default maximum heap size is set to 60% of the maximum amount of memory available. The maximum amount of memory available in this case is the minimum of the physical amount of memory on the machine and the total amount of memory available to the process.
5.10 Other JVM alternatives

It would be good to perform the tests also with another JVM. Unfortunately the author did not find any other JVM for the Tru64 UNIX Alpha platform.

There however exists a Java native compiler for Tru64, called TowerJ ([29]). A native compiler could be used, although the important Bytecode platform compatibility will then be lost (only the Java source code is still platform compatible). The ASA backwards compatibility is lost both when using Bytecode and a native compiler.

TowerJ makes optimizations specific to the Tru64 UNIX Alpha platform during compile time and then creates a machine code executable rather than class files. This might give increased performance. It would have been interesting to test TowerJ. But TowerJ could not be tested in this thesis since there currently does not exist any evaluation version for Tru64.

5.10.1 Benchmark

It is hard to say much about the performance of the Compaq Fast VM since no other JVM is available for measurement on Tru64. But according to the test results in the SPECjbb2000 Java benchmark (for more information about the test see reference [28]) Compaq Fast VM is the leader in server-side performance in for instance the two processors category.

When looking at the number of operations per second, the Compaq Fast VM outperforms the competitors Hewlett-Packard and IBM with 87% and 21% respectively. This does not really prove anything, but gives a good guidance to that the Compaq Fast VM is a quite good JVM.

5.11 Results

Figures 42 to 49 and Table 11 show the results from the different test cases measured under JVM and APZ VM.

The test runs were performed using the Fast 64 VM together with the copying garbage collector.
FIGURE 42. Test case 1 measured with JVM (diamond) and APZ VM (triangle)

FIGURE 43. Test case 2 measured with JVM (diamond) and APZ VM (triangle)
Figure 44. Test case 3 measured with JVM (diamond) and APZ VM (triangle)

Figure 45. Test case 4 measured with JVM (diamond) and APZ VM (triangle)
FIGURE 46. Test case 5 measured with JVM (diamond) and APZ VM (triangle)

FIGURE 47. Test case 6 measured with JVM (diamond) and APZ VM (triangle)
5.12 Analysis

The diagrams show that the JVM system generally executes at least at approximately 50% of the speed of the APZ VM system. It takes some more time for the JVM to warm up and the test values are less stable than the APZ VM. Table 7 shows the JVM speed compared to the APZ VM speed for each individual test case.
Here follow some short comments on the diagrams for the specific test cases.

- **Test case 1.** The JVM is quite slow in the beginning starting out with 58% of the APZ VM speed. It then catches up somewhat and lies quite stable around 69% and at most at 71% of the APZ VM speed.

- **Test case 2.** The JVM shows the value of 43% in the first test run. It then lies around 63% but unexpectedly ends with a value of 56%. Because of the low value of the call-parameter, this test case has the shortest measured times of them all. This makes it quite unreliable with larger chances of random influences. This might be the reason for the two lower values presented by the JVM. But the warm up time is also more noticeable with short test cases. In longer test cases the warm up time is small compared to the length of the test, so the JVM has time to catch up before the actual time is measured.

- **Test case 3.** The JVM performs very well with values ranging from 80% to 95% of the APZ VM speed. The values from the JVM are however very unstable. This test case has no intermediate blocks and takes very short time to perform. As in test case 2, this makes this test case somewhat unreliable and therefore probably also unstable. Not much attention is therefore paid to the high results of this test case. Since no intermediate blocks are used, the active code volume is thereby very small compared to the other test cases. If this test case had been more reliable, it could have indicated that the run-time environment part of the JVM system is fast. Or it could have indicated that the JVM or APZ VM is slow compared to the other for larger tests and smaller tests respectively.

- **Test case 4.** This test case has the second longest time to complete, which makes it reliable and more stable. The APZ VM is very stable the whole time. The JVM has a small warm up time which is 64% of the APZ VM speed, but then lies stable.
around 66% of the APZ VM speed. Since the sigs-parameter is set to 0 the number of sent signals is lowered.

- **Test case 5.** This test case is like test case 1, but the inddata-parameter is set to 0. The measured values are also very similar. The JVM starts with 57% and then lies around 68% of the APZ VM speed. The measurements have insignificant differences from test case 1. The accessing of the inddata-variables seems to have approximately equal influence on both the JVM and the APZ VM.

- **Test case 6.** This test case is like test case 1 and 5, but the inddata-parameter is set to 6. The JVM starts with 57% and then lies around 67% of the APZ VM speed. Again the JVM seems to handle the inddata-variable access approximately equal to the APZ VM.

- **Test case 7.** Differs from test case 1 in that the conts-parameter has a larger value. The JVM starts with 58% and then lies around 66% of the APZ VM speed. The differences in measurements from test case 1 are very small. This can be a result of the more signals that have to be sent. It might indicate that the JVM is slower than then APZ VM in signal handling, which it is expected to be.

- **Test case 8.** This is the most complete and reliable test case. It resembles EB1 and takes the longest time to complete. The JVM warms up slower than the APZ VM and lies on 66% of its speed. It then lies quite stable on 68% of its speed.

The conclusions from all the tests is that the JVM generally is slower, takes longer to warm up, and is more unstable than the APZ VM.

The JVM often executes in over 60% of the speed of the APZ VM, a speed of 50% is however considered a more reliable value. This is a very high and unexpected value. The JVM might however get speed problems with larger tests or when the whole run-time environment is implemented. The extra JVM layer for the run-time environment however does not seem to slow down as much as expected.

The warm up time probably depends on the implementation of the JIT compiler in the JVM. It seems to demand more time than the JIT compiler in the APZ VM to warm up.

It is hard to tell why the JVM is more unstable than the APZ VM. But it might be a result of the garbage collector. It could also be that the performance of the JVM is more easily affected by other processes that executed on the test system concurrently during the test.
Since the *tested* APZ VM version is several times faster than the APZ, the tests indicate that the JVM system is a few times faster than the APZ. The *current* version of the APZ VM is however several times faster than the JVM system. The *finished* APZ VM version will be even faster.

The requirements of the JVM system for it to be interesting for Ericsson, is that it should have a speed of approximately at least 70% of the *finished* version of the APZ VM. This is a very high requirement, which means that the JVM currently is too slow to be interesting.

The expectations on the JVM system were generally bad, probably since the JVM is known for its supposed bad performance. The results were much better than expected and the JVM system is generally feasible, but not immediately applicable with the current requirements.
6 Results

This chapter presents the results from the thesis work.

6.1 Limitations

This section shows the major limitations of the thesis work compared to the real system. The limitation in speed is clearly the most relevant of them.

6.1.1 Speed

The speed of the JVM system lies below the requirements from Ericsson. This is mostly because of the extra layer that is provided by the JVM. The JVM system however gains speed since it is limited.

6.1.2 Garbage collector

The fact that the JVM run-time environment uses a garbage collector destroys some of the platform compatibility. That is because garbage collectors often do not work in the same way for every JVM. Even though the garbage collector seems to work in a sufficient way for the JVM tested in the thesis, this does not mean that it does work that good and in the same way for all other JVMs.

6.1.3 Emulation problems

Since the run-time environment not does an emulation of the APZ, some old applications might not work as they should. If applications intentionally or unintentionally have made use of side-effects of the way Plex is implemented on the APZ, they may not work properly on the run-time environment. If for example something is placed in a temporary register, it still might be there later during execution (but should not be trusted to be) on the APZ. The JVM run-time environment does not implement the temporary registers in the same way, and the results will not be similar. This is hopefully a small problem, since well implemented applications never should make use of these kinds of side-effects.

6.1.4 No ASA support

Another problem is that inline ASA code is not supported by the JVM run-time environment. Inline ASA code is sometimes used in time critical Plex code. It would probably not be possible to emulate or translate the ASA code in an effective way since it differs much from Bytecode. A rewrite of the source code from ASA to Plex is probably a better approach in the specific cases.
6.1.5 Goto-statement
In difference to Bytecode, the Plex language fully supports the goto statement. In many cases the goto statements can be compiled to Bytecode, but not always. It depends on what way the programmer has used it. Bytecode only supports a limited form of the goto statement.

6.2 Advantages
The JVM run-time environment has the big advantage of being processor independent. All applications can be moved to another platform that has a good JVM implementation without any need to recompile or rewrite the source code.

6.3 Conclusions
The thesis work shows that the JVM currently is not suitable to be used as a replacement for the currently used AXE processor architecture. Future improvements in JVM technology may however change this. See for example the development of new garbage collecting technology that is described in reference [33].

Most parts of the system can be implemented to run on the JVM and the performance is good compared to the APZ. The APZ VM is however faster and the performance is not good enough according to the Ericsson requirements.

The author is satisfied with the thesis work in that it fulfilled its purpose and that more than the planned work was completed. Although the resulting system did not become sufficiently fast for large AXE exchanges, it is still much faster than what was expected.

6.4 Further research
Further research includes the construction of a Plex to Bytecode compiler, testing of larger applications, and completion of the system.

6.4.1 Plex to Bytecode compiler
A Plex to Bytecode compiler would make the system much more useful and easier to test. Compilation by hand of large test applications takes too much time.
6.4.2 Testing
There are many aspects of the system that need to be tested further. The most important ones are probably the speed and the garbage collection. For larger applications the compacting garbage collector would probably have to be used instead of the copying garbage collector. This is to reduce the pause times. Observe that the garbage collectors do not work in the same way for every JVM.

A more complete test then the EB test can also be used to test the system more thoroughly.

6.4.3 Completion
The AXE processor architecture and the Plex language are too big to be fully covered in the thesis. Thereby not all parts are implemented and tested.

6.5 Usage
The JVM run-time environment is currently not useful by itself. But if it is complemented with a Plex to Bytecode compiler it might have use as a portable test platform.
## Appendix A  Javadoc

### Package

**plex.runtime**

### Class Summary

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Represents the behaviour that must be implemented by every Plex block.</td>
</tr>
<tr>
<td>Signal</td>
<td>Represents the behaviour that must be implemented by every Plex signal.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClockInterrupt</td>
<td>Simulates the CIS (Clock Interrupt Signal).</td>
</tr>
<tr>
<td>GregorianCalendarExtended</td>
<td>Implements the java.util.GregorianCalendar, and makes sure that the millisecond methods not are protected.</td>
</tr>
<tr>
<td>JobBuffer</td>
<td>Implements a job buffer.</td>
</tr>
<tr>
<td>JobHandler</td>
<td>Handles the execution of jobs.</td>
</tr>
<tr>
<td>JobTable</td>
<td>Implements the Job table that is used by Plex.</td>
</tr>
<tr>
<td>Main</td>
<td>Starts the run-time environment.</td>
</tr>
<tr>
<td>TimeQueueAbsolute</td>
<td>Implements the absolute time queue.</td>
</tr>
<tr>
<td>TimeQueueRelative</td>
<td>Implements a relative time queue.</td>
</tr>
</tbody>
</table>
plex.runtime

Block

Declaration
public interface Block

Description
Represents the behaviour that must be implemented by every Plex block.

### Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void clear()</td>
</tr>
<tr>
<td>public void dump(FileOutputStream)</td>
</tr>
<tr>
<td>public void enter(Signal)</td>
</tr>
<tr>
<td>public void reload(FileInputStream)</td>
</tr>
<tr>
<td>public boolean setDump(String, boolean)</td>
</tr>
</tbody>
</table>

### Methods

**clear()**

```java
public void clear()
```

Clears all variables that have the CLEAR property.

**dump(FileOutputStream)**

```java
public void dump(java.io.FileOutputStream out)
```

Dumps all variables that have the DUMP property.

**Parameters:**
- `out` - Output file to write the values of the variables.

**enter(Signal)**

```java
public void enter(Signal signal)
```

Receives every signal that are sent to the block. The signals are delegated to their associated signal handler.

**Parameters:**
- `signal` - The sent signal.
reload(FileInputStream)

public void reload(java.io.FileInputStream in)

Reloads all variables that have the RELOAD property.

Parameters:

   in - Input file that contains the values that will be assigned to the variables.

setDump(String, boolean)

public boolean setDump(java.lang.String name, boolean status)

Sets or clears the DUMP property of the specified variable.

Parameters:

   name - The variable name.

   status - The DUMP status of the variable (true is set, and false is cleared).

Returns: True if specified variable exists, otherwise false.
plex.runtime

ClockInterrupt

Declaration

```java
class ClockInterrupt extends java.util.TimerTask
```

All Implemented Interfaces: java.lang.Runnable

Description

Simulates the CIS (Clock Interrupt Signal). Sends an interrupt every 10 ms.

Member Summary

| Constructors | ClockInterrupt(JobTable) Constructs the clock interrupt. |
| Methods | public void run() Starts sending the clock interrupts. |

Inherited Member Summary

Fields inherited from class java.util.TimerTask
CANCELLED, EXECUTED, SCHEDULED, VIRGIN, lock, nextExecutionTime, period, state

Methods inherited from class java.lang.Object
<clinit>, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from class java.util.TimerTask
cancel, scheduledExecutionTime

Constructors

ClockInterrupt(JobTable)

ClockInterrupt(JobTable jobTable)
Constructs the clock interrupt.

**Parameters:**

- `jobTable` - Reference to the job table.

## Methods

### run()

```java
public void run()
```

Starts sending the clock interrupts. Clock interrupts are sent to the job table every 10 ms.

**Overrides:** `java.util.TimerTask.run()` in class `java.util.TimerTask`
plex.runtime
GregorianCalendarExtended

**Declaration**

class GregorianCalendarExtended extends java.util.GregorianCalendar

java.lang.Object
|  --> java.util.Calendar
    |  --> java.util.GregorianCalendar
        |  --> plex.runtime.GregorianCalendarExtended

**All Implemented Interfaces:** java.lang.Cloneable, java.io.Serializable

**Description**

Implements the java.util.GregorianCalendar, and makes sure that the millisecond methods not are protected. The API for Java 1.3.1 specifies the millisecond methods with protected access. To be able to use the run-time environment effectively under Java 1.3.1 (which currently are the latest JRE from Compaq) these methods must not be protected. This class overrides them and makes them public.

**Member Summary**

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GregorianCalendarExtended()</td>
</tr>
<tr>
<td>Default constructor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public long getTimeInMillis()</td>
</tr>
<tr>
<td>Gets this Calendar’s current time as a long.</td>
</tr>
<tr>
<td>public void setTimeInMillis(long)</td>
</tr>
<tr>
<td>Sets this Calendar’s current time from the given long value.</td>
</tr>
</tbody>
</table>

**Inherited Member Summary**

**Fields inherited from class java.util.Calendar**

AM, AM_PM, APRIL, AUGUST, DATE, DAY_OF_MONTH, DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_YEAR, DECEMBER, DST_OFFSET, ERA, FEBRUARY, FIELD_COUNT, FRIDAY, HOUR, HOUR_OF_DAY, INTERNALLY_SET, JANUARY, JULY, JUNE, MARCH, MAY, MILLISECOND, MINIMUM_USER_STAMP, MINUTE, MONDAY, MONTH, NOVEMBER, OCTOBER, PM, SATURDAY, SECOND, SEPTEMBER, SUNDAY, THURSDAY, TUESDAY, UNDECEMBER, UNSET, WEDNESDAY, WEEK_OF_MONTH, WEEK_OF_YEAR, YEAR, ZONE_OFFSET, areAllFieldsSet, areFieldsSet, currentSerialVersion, fields, isSet, isTimeSet, stamp, time

**Fields inherited from class java.util.GregorianCalendar**
Inherited Member Summary

AD, BC, serialVersionUID

Methods inherited from class java.util.Calendar

after, before, clear, clear, clone, complete, get, getAvailableLocales, getFirstDayOfWeek, get Instance, get Instance, get Instance, get Instance, getMinimalDaysInFirstWeek, getTime, getTimeZone, internalGet, internalSet, isLenient, isSet, set, set, set, set, setFirstDayOfWeek, setLenient, setMinimalDaysInFirstWeek, setTime, setTimeZone, toString

Methods inherited from class java.util.GregorianCalendar

<clinit>, add, computeFields, computeTime, equals, getActualMaximum, getActualMinimum, getGreatestMinimum, getGregorianChange, getISOYear, getLeastMaximum, getMaximum, getMinimum, hashCode, inDaylightTime, isLeapYear, roll, roll, setGregorianChange

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Constructors

GregorianCalendarExtended()

GregorianCalendarExtended()

Default constructor.

Methods

getTimeInMillis()

public long getTimeInMillis()

Gets this Calendar’s current time as a long.

Overrides: java.util.Calendar.getTimeInMillis() in class java.util.Calendar

Returns: the current time as UTC milliseconds from the epoch.

setTimeInMillis(long)

public void setTimeInMillis(long millis)

Sets this Calendar’s current time from the given long value.

Overrides: java.util.Calendar.setTimeInMillis(long) in class java.util.Calendar

Parameters:
millis - The new time in UTC milliseconds from the epoch.
plex.runtime

JobBuffer

Declaration

class JobBuffer

description

Implements a job buffer.

Member Summary

Constructors

JobBuffer(byte, int)

Creates a job buffer.

Methods

synchronized Signal front()

Removes the front from the queue and returns the stored signal.

synchronized boolean queue(Signal)

Queues a signal in this job buffer.

public String toString()

Returns a string representation of the job buffer.

Inherited Member Summary

Methods inherited from class java.lang.Object

<clinit>, clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait

Constructors

JobBuffer(byte, int)

JobBuffer(byte level, int size)

Creates a job buffer.

Parameters:

level - The job buffer priority level.

size - The size of the queue.
### Methods

**front()**

```java
synchronized Signal front()
```

Removes the front from the queue and returns the stored signal.

**Returns:** The signal that was stored in the front of the queue.

**queue(Signal)**

```java
synchronized boolean queue(Signal signal)
```

Queues a signal in this job buffer. If the queue was full, an error message is printed to standard error. The returnvalue specifies if the operation was successful or not.

**Parameters:**
- `signal` - The signal to be queued.

**Returns:** False if the operation failed because of a full queue, otherwise true.

**toString()**

```java
public java.lang.String toString()
```

Returns a string representation of the job buffer. This includes the job level, size, front index, and back index.

**Overrides:** `java.lang.Object.toString()` in class `java.lang.Object`

**Returns:** A string representation of the job buffer.
**plex.runtime**

**JobHandler**

**Declaration**

```java
public class JobHandler
```

**Description**

Handles the execution of jobs. Creates the different queues and schedules the jobs. Represents the visual interface for the run-time environment.

**Member Summary**

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static final asserts</td>
<td>synchronized void clearInterrupt(byte)</td>
</tr>
<tr>
<td>Makes “asserts” appear or disappear in the compiled class-files.</td>
<td>Clears the interrupt flag for the specified level.</td>
</tr>
<tr>
<td>public static final JBA</td>
<td>public boolean jobTableInsert(Signal, int)</td>
</tr>
<tr>
<td>Constant for addressing job buffer A.</td>
<td>Inserts a signal into the job table.</td>
</tr>
<tr>
<td>public static final JBB</td>
<td>public boolean jobTableRemove(Signal)</td>
</tr>
<tr>
<td>Constant for addressing job buffer B.</td>
<td>Deletes a signal from the job table.</td>
</tr>
<tr>
<td>public static final JBC</td>
<td>public boolean jobTableSetInterval(Signal, int)</td>
</tr>
<tr>
<td>Constant for addressing job buffer C.</td>
<td>Updates the interval of a signal placed in the job table.</td>
</tr>
<tr>
<td>public static final JBD</td>
<td>public boolean queueAbsolute(byte, Signal, byte, byte, byte, byte)</td>
</tr>
<tr>
<td>Constant for addressing job buffer D.</td>
<td>Queues a signal in the absolute time queue.</td>
</tr>
<tr>
<td>static final JOB_TABLE</td>
<td>public boolean queueExternal(byte, Signal)</td>
</tr>
<tr>
<td>Constant for addressing the job table.</td>
<td>Queues a RP-CP signal in the specified job buffer.</td>
</tr>
<tr>
<td>public static final jobHandler</td>
<td></td>
</tr>
<tr>
<td>Reference to the only instance of the job handler (Singleton design pattern).</td>
<td></td>
</tr>
</tbody>
</table>
**Member Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean queueInternal(byte, Signal)</td>
<td>Queues a CP-CP signal in the specified job buffer.</td>
</tr>
<tr>
<td>public boolean queueRelative(byte, Signal, byte, int)</td>
<td>Queues a signal in one of the relative time queues.</td>
</tr>
<tr>
<td>synchronized void setInterrupt(byte)</td>
<td>Sets the interrupt flag for the specified level.</td>
</tr>
<tr>
<td>void start()</td>
<td>This method goes in an infinite loop checking for interrupts.</td>
</tr>
<tr>
<td>public void yield()</td>
<td>Checks if any higher priority interrupts have occurred.</td>
</tr>
</tbody>
</table>

**Inherited Member Summary**

Methods inherited from class java.lang.Object

- <clinit>, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

**Fields**

asserts

public static final boolean asserts

Makes “asserts” appear or disappear in the compiled class-files. True means that asserts are included, while false eliminates them.

JBA

public static final byte JBA

Constant for addressing job buffer A.

JBB

public static final byte JBB

Constant for addressing job buffer B.

JBC

public static final byte JBC

Constant for addressing job buffer C.

JBD

public static final byte JBD

Constant for addressing job buffer D.
JOB_TABLE

    static final byte JOB_TABLE
    Constant for addressing the job table.

jobHandler

    public static final JobHandler jobHandler
    Reference to the only instance of the job handler (Singleton design pattern).

TQB

    public static final byte TQB
    Constant for addressing time queue B.

TQC

    public static final byte TQC
    Constant for addressing time queue C.

TQD

    public static final byte TQD
    Constant for addressing time queue D.

Methods

clearInterrupt(byte)

    synchronized void clearInterrupt(byte level)
    Clears the interrupt flag for the specified level.
    Parameters:
    level - The interrupt level (JOB_TABLE, JBA, JBB, JBC, or JBD).

jobTableInsert(Signal, int)

    public boolean jobTableInsert(Signal signal, int interval)
    Inserts a signal into the job table.
    Parameters:
    signal - The signal to be queued.
    interval - The job interval of the signal. This is specified in number of 10 ms intervals (number of CIS interrupts).
    Returns: False if the operation failed because of a full table, otherwise true.
    Throws:
    NullPointerException - if signal was not specified.
    IllegalArgumentException - if specified interval can not be handled by the job table.
jobTableRemove(Signal)

```java
public boolean jobTableRemove(Signal signal)
```

Deletes a signal from the job table.

**Parameters:**
- `signal` - The signal to be deleted.

**Returns:** False if the operation failed because the signal was not part of the table, otherwise true.

**Throws:**
- `NullPointerException`- if signal was not specified.

---

jobTableSetInterval(Signal, int)

```java
public boolean jobTableSetInterval(Signal signal, int interval)
```

Updates the interval of a signal placed in the job table.

**Parameters:**
- `signal` - The signal to be updated.
- `interval` - The job interval of the signal. This is specified in number of 10 ms intervals.

**Returns:** False if the operation failed because the signal was not part of the table, otherwise true.

**Throws:**
- `NullPointerException`- if signal was not specified.
- `IllegalArgumentException`- if specified interval can not be handled by the job table.

---

queueAbsolute(byte, Signal, byte, byte, byte, byte)

```java
public boolean queueAbsolute(byte level, Signal signal, byte month, byte day, byte hour, byte minute)
```

Queues a signal in the absolute time queue. The return value specifies if the operation was successful or not.

**Parameters:**
- `level` - Priority level of the signal (JBA, JBB, JBC, or JBD).
- `signal` - The signal to be queued.
- `month` - The month when the signal will be sent. Zero means sending each month that has specified day.
- `day` - The day when the signal will be sent. Zero means sending each day.
- `hour` - The hour when the signal will be sent.
- `minute` - The minute when the signal will be sent.

**Returns:** false if the absolute time queue is full, otherwise true.

**Throws:**
- `IndexOutOfBoundsException`- if specified level is illegal.
- `IllegalArgumentException`- if specified month, day, hour, or minute is illegal.
- `NullPointerException`- if signal was not specified.

---

queueExternal(byte, Signal)

```java
public boolean queueExternal(byte level, Signal signal)
```
Javadoc

Queues a RP-CP signal in the specified job buffer. This method is intended for external signals (RP-CP).

**Parameters:**
- `level` - Level of the job buffer (JBA, JBB, JBC, or JBD).
- `signal` - The signal to be queued.

**Returns:** False if the operation failed because of a full queue, otherwise true.

**Throws:**
- `IndexOutOfBoundsException` - if specified level is illegal.
- `NullPointerException` - if signal was not specified.

```java
public boolean queueInternal(byte level, Signal signal)
```

Queues a CP-CP signal in the specified job buffer. This method is intended for internal signals (CP-CP). If the currently executing job belongs to a priority level that is lower than the priority of the queued signal, that job is interrupted.

**Parameters:**
- `level` - Level of the job buffer (JBA, JBB, JBC, or JBD).
- `signal` - The signal to be queued.

**Returns:** False if the operation failed because of a full queue, otherwise true.

**Throws:**
- `IndexOutOfBoundsException` - if specified level is illegal.
- `NullPointerException` - if signal was not specified.

```java
public boolean queueRelative(byte level, Signal signal, byte queue, int interval)
```

Queues a signal in one of the relative time queues. The returnvalue specifies if the operation was successful or not.

**Parameters:**
- `level` - Priority level of the signal (JBA, JBB, JBC, or JBD).
- `signal` - The signal to be queued.
- `queue` - Which relative time queue to use. Allowed values are TQB, TQC, and TQD.
- `interval` - The number of times the internal time interval of the queue should be decreased before placing the signal in the specified job buffer.

**Returns:** false if the absolute time queue is full, otherwise true.

**Throws:**
- `IndexOutOfBoundsException` - if specified level is illegal.
- `IllegalArgumentException` - if specified queue does not exist or specified interval can not be handled by the specified queue.
- `NullPointerException` - if signal was not specified.

```java
synchronized void setInterrupt(byte level)
```
Sets the interrupt flag for the specified level.

**Parameters:**

level - The interrupt level (JOB_TABLE, JBA, JBB, JBC, or JBD).

**start()**

```java
void start()
```

This method goes in an infinite loop checking for interrupts. When interrupts are encountered the corresponding signal is sent.

**yield()**

```java
public void yield()
```

Checks if any higher priority interrupts have occurred. If there exist any jobs with higher priority than the currently executing job, those jobs are executed. Since jobs in the job table and job buffer A or B can not be interrupted, this method is only invoked by C and D level jobs. The purpose of this method is to give higher priority jobs a chance to execute.
plex.runtime

JobTable

Declaration

class JobTable

java.lang.Object
|-- plex.runtime.JobTable

Description

Implements the Job table that is used by Plex. The class is also responsible for creating the time queues and insert them into the table.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static final MAX_INTERVAL</td>
</tr>
<tr>
<td>The largest interval allowed in job table.</td>
</tr>
<tr>
<td>static final MIN_INTERVAL</td>
</tr>
<tr>
<td>The smallest interval allowed in job table.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobTable()</td>
</tr>
<tr>
<td>Creates the Job table.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronized void cis()</td>
</tr>
<tr>
<td>This method is called when a CIS interrupt occurs.</td>
</tr>
<tr>
<td>boolean insert(Signal, int)</td>
</tr>
<tr>
<td>Inserts a signal into the Job table.</td>
</tr>
<tr>
<td>boolean queueAbsolute(byte, Signal, byte, byte, byte, byte)</td>
</tr>
<tr>
<td>Queues a signal in the absolute time queue.</td>
</tr>
<tr>
<td>boolean queueRelative(byte, Signal, byte, byte, int)</td>
</tr>
<tr>
<td>Queues a signal in a relative time queue.</td>
</tr>
<tr>
<td>boolean remove(Signal)</td>
</tr>
<tr>
<td>Removes a signal from the Job table.</td>
</tr>
<tr>
<td>synchronized void scan()</td>
</tr>
<tr>
<td>Scans the Job table.</td>
</tr>
<tr>
<td>boolean setInterval(Signal, int)</td>
</tr>
<tr>
<td>Updates the interval of a signal.</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
<tr>
<td>Returns a string representation of the job table.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class java.lang.Object
Fields

MAX_INTERVAL

    static final int MAX_INTERVAL
    The largest interval allowed in job table.

MIN_INTERVAL

    static final int MIN_INTERVAL
    The smallest interval allowed in job table.

Constructors

JobTable()

    JobTable()
    Creates the Job table. Also creates the time queues and inserts them into the table.

Methods

cis()

    synchronized void cis()
    This method is called when a CIS interrupt occurs. Increases the number of interrupts and sets the interrupt
tag in the job handler if the counter was zero.

insert(Signal, int)

    boolean insert(Signal signal, int interval)
    Inserts a signal into the Job table. If the table was full, an error message is printed to standard error. The
    returnvalue specifies if the table was full or not.

Parameters:
    signal - The signal to be queued.
    interval - The time the job table should wait before sending the signal. This is specified in number
    of 10 ms intervals.

Returns: False if the operation failed because of a full table, otherwise true.

queueAbsolute(byte, Signal, byte, byte, byte, byte)

    boolean queueAbsolute(byte level, Signal signal, byte month, byte day, byte hour, byte minute)
Queues a signal in the absolute time queue. The returnvalue specifies if the operation was successful or not.

**Parameters:**
- `level` - The job buffer level of the signal. Allowed values are JobHandler.JBA, JobHandler.JBB, JobHandler.JBC, and JobHandler.JBD.
- `signal` - The signal to be queued.
- `month` - The month when the signal will be sent. Zero means sending each month that has the specified day.
- `day` - The day when the signal will be sent. Zero means sending each day.
- `hour` - The hour when the signal will be sent.
- `minute` - The minute when the signal will be sent.

**Returns:** False if the specified time queue does not exist, otherwise true.

```
queueRelative(byte, Signal, byte, int)
```

boolean `queueRelative(byte level, Signal signal, byte queue, int interval)`

Queues a signal in a relative time queue. The returnvalue specifies if the operation was successful or not.

**Parameters:**
- `level` - The job buffer level of the signal. Allowed values are JobHandler.JBA, JobHandler.JBB, JobHandler.JBC, and JobHandler.JBD.
- `signal` - The signal to be queued.
- `queue` - Which relative time queue to use. Allowed values are JobHandler.TQB, JobHandler.TQC, and JobHandler.TQD.
- `interval` - The number of times the internal time interval of the queue should be decreased before placing the signal in the specified job buffer.

**Returns:** False if the time queue already was full, otherwise true.

```
remove(Signal)
```

boolean `remove(Signal signal)`

Removes a signal from the Job table. If the signal could not be removed, an error message is printed to standard error. The returnvalue specifies if the operation was successful or not.

**Parameters:**
- `signal` - The signal to be queued.

**Returns:** False if the operation failed because the signal was not part of the table, otherwise true.

```
scan()
```

synchronized void `scan()`

Scans the Job table. Increases the internal clock and sends the signals that are scheduled at that time.

```
setInterval(Signal, int)
```

boolean `setInterval(Signal signal, int interval)`

Updates the interval of a signal. If the signal interval could not be changed, an error message is printed to standard error. The returnvalue specifies if the operation was successful or not.
Parameters:

- **signal**: The signal to be updated.
- **interval**: The time the job table should wait before sending the signal. This is specified in number of 10 ms intervals.

Returns: False if the operation failed because the signal was not part of the table, otherwise true.

toString()

```java
public java.lang.String toString()
```

Returns a string representation of the job table. This includes the start index of the signal list, free list, and the inactive list. This is followed by the internal counter. It also includes the whole table with index, interval, and pointer for each row.

Overrides: `java.lang.Object.toString()` in class `java.lang.Object`

Returns: A string representation of the job table.
plex.runtime
Main

Declaration
class Main

java.lang.Object
|--plex.runtime.Main

Description
Starts the run-time environment. Creates the job handler and sends initial signals.

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main()</td>
<td>Default constructor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void main(String[])</td>
<td>Creates the job handler and sends initial signals.</td>
</tr>
</tbody>
</table>

### Inherited Member Summary

Methods inherited from class java.lang.Object
- <clinit>, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait

### Constructors

Main()

Main()
Default constructor.

### Methods

main(String[])

public static void main(java.lang.String[] args)
Creates the job handler and sends initial signals.
Parameters:
  \texttt{args} - Input parameters to the run-time environment (currently not used).
plex.runtime

Signal

Declaration
public interface Signal

All Known Implementing Classes: TimeQueueAbsolute, TimeQueueRelative

Description
Represents the behaviour that must be implemented by every Plex signal.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void send()</td>
</tr>
<tr>
<td>Sends this signal to the block that was specified during construction of the signal.</td>
</tr>
</tbody>
</table>

Methods

send()

public void send()

Sends this signal to the block that was specified during construction of the signal.
plex.runtime

TimeQueueAbsolute

Declaration

```java
class TimeQueueAbsolute implements plex.runtime.Signal
java.lang.Object
    |   -- plex.runtime.TimeQueueAbsolute
```

All Implemented Interfaces: Signal

Description

Implements the absolute time queue. The queue is implemented as a table.

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeQueueAbsolute(JobTable, int, int)</td>
</tr>
<tr>
<td>Constructs the absolute time queue.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean queue(byte, Signal, byte, byte, byte, byte)</td>
</tr>
<tr>
<td>Queues a signal in the absolute time queue.</td>
</tr>
<tr>
<td>public void send()</td>
</tr>
<tr>
<td>Checks through the queue and places jobs that are scheduled at the current time in their associated job buffer.</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
<tr>
<td>Returns a string representation of the absolute time queue.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

| Methods inherited from class java.lang.Object |
| <clinit>, clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait |

### Constructors

**TimeQueueAbsolute(JobTable, int, int)**

*TimeQueueAbsolute(JobTable jobTable, int size, int interval)*

Constructs the absolute time queue.
Parameters:

- jobTable - Reference to the job table.
- size - The size of the queue.
- interval - The interval (specified in number of CIS interrupts) between checks of the queue.

Methods

queue(byte, Signal, byte, byte, byte, byte)

boolean queue(byte level, Signal signal, byte month, byte day, byte hour, byte minute)

Queues a signal in the absolute time queue. If the queue was full or if specified date is invalid, an error message is printed to standard error. The return value specifies if the operation was successful or not.

Parameters:

- level - Priority level of the signal (JBA, JBB, JBC, or JBD).
- signal - The signal to be queued.
- month - The month when the signal will be sent. Zero means sending each month that has the specified day.
- day - The day when the signal will be sent. Zero means sending each day.
- hour - The hour when the signal will be sent.
- minute - The minute when the signal will be sent.

Returns: false if the operation failed, otherwise true.

send()

public void send()

Checks through the queue and places jobs that are scheduled at the current time in their associated job buffer. All jobs that match the current date (or earlier) are placed in their associated job buffer.

Specified By: send() in interface TimeQueueAbsolute

toString()

public java.lang.String toString()

Returns a string representation of the absolute time queue. This includes the start index of the signal list and the free list. This is followed by the current system time. It also includes the whole table with index, pointer, and date for each row.

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: A string representation of the absolute time queue.
plex.runtime

TimeQueueRelative

Declaration

class TimeQueueRelative implements plex.runtime.Signal

java.lang.Object
|-- plex.runtime.TimeQueueRelative

All Implemented Interfaces: Signal

Description

Implements a relative time queue. The queue is implemented as a table.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static final TQB_MAX_INTERVAL</td>
</tr>
<tr>
<td>static final TQB_MIN_INTERVAL</td>
</tr>
<tr>
<td>static final TQC_MAX_INTERVAL</td>
</tr>
<tr>
<td>static final TQC_MIN_INTERVAL</td>
</tr>
<tr>
<td>static final TQD_MAX_INTERVAL</td>
</tr>
<tr>
<td>static final TQD_MIN_INTERVAL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeQueueRelative(JobTable, int, int)</td>
</tr>
<tr>
<td>Constructs a relative time queue.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean queue(byte, Signal, int)</td>
</tr>
<tr>
<td>Queues a signal in the relative time queue.</td>
</tr>
<tr>
<td>public void send()</td>
</tr>
<tr>
<td>Checks through the queue and eventually places jobs in the job buffers.</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
<tr>
<td>Returns a string representation of the relative time queue.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class java.lang.Object
Fields

TQB_MAX_INTERVAL
static final int TQB_MAX_INTERVAL
Time queue B max interval.

TQB_MIN_INTERVAL
static final int TQB_MIN_INTERVAL
Time queue B min interval.

TQC_MAX_INTERVAL
static final int TQC_MAX_INTERVAL
Time queue C max interval.

TQC_MIN_INTERVAL
static final int TQC_MIN_INTERVAL
Time queue C min interval.

TQD_MAX_INTERVAL
static final int TQD_MAX_INTERVAL
Time queue D max interval.

TQD_MIN_INTERVAL
static final int TQD_MIN_INTERVAL
Time queue D min interval.

Constructors

TimeQueueRelative(JobTable, int, int)
TimeQueueRelative(JobTable jobTable, int size, int interval)
Constructs a relative time queue.

Parameters:
jobTable - Reference to the job table.
size - The size of the queue.
interval - The interval (specified in number of CIS interrupts) between checks of the queue.
Methods

queue(byte, Signal, int)

    boolean queue(byte level, Signal signal, int interval)

Queues a signal in the relative time queue. If the queue was full, an error message is printed to standard error. The returnvalue specifies if the operation was successful or not.

Parameters:
    level - The job buffer level of the signal.
    signal - The signal to be queued.
    interval - The number of times the internal time interval of the queue should be decreased before placing the signal in the specified job buffer.

Returns: False if the time queue already was full, otherwise true.

send()

    public void send()

Checks through the queue and eventually places jobs in the job buffers. All job counters are decreased and when reaching zero they are placed in their specified job buffer.

Specified By: send() in interface TimeQueueRelative

toString()

    public java.lang.String toString()

Returns a string representation of the relative time queue. This includes the queue interval (measured in CIS). It also includes the whole table with index, level, and interval for each row.

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: A string representation of the relative time queue.
References


References


Index

A
absolute time queue 33
addressing 40
APZ 8, 58
APZ VM 11, 31, 58
Array 15
ASA 2, 7
AXE 7
B
BAL 1 27
BAL 2 27
Base Address Table 19
Base level 1 27
Base level 2 27
BAT 19
Block 79
block 13, 35, 38
Block Number Receiving 21
BN-R 21, 25
BUFFER 17, 46
buffered signal 23, 58
C
call table 89
Central processor 7, 8, 77
CIS 27
CIS thread 74
class descriptions 78
class diagram 78
classloader 36
CLEAR 18, 47
Clock Interrupt Signal 27
ClockInterrupt 80
combined signal 23
common variables 20
Compacting collector 97
Compaq Fast VM 94
Copying collector 97
Copying garbage collection 97
CP 7
CP-CP signals 24
CP-RP signals 24
D
data encapsulation 13
data sector 14, 38
Index

Data Store 9, 10
deadlock 58
deadlock analysis 73
declare sector 14, 38
destination table 89
direct signal 22, 59
distributed RP table 24
DS 9, 17, 46
DUMP 18, 47
E
EB 83
EBFIRST 85
EBNXT 85
EMRP 7
Emulator Benchmark 83
Extension Module Regional Processor 7
F
Field variable 15, 42
file 14, 19, 49
File size alteration 20
fixed priority scheduling 62
free store 99
function change 19, 37, 49
function pointers 36
G
Garbage collection 96
Global Signal Distribution Table 25
Global Signal Number 25
GregorianCalendarExtended 80
GSDT 25
GSN 25
H
heap 99
Heap management 100
I
IA 25
ID sector 14
indirect garbage collectors 97
individual variables 20
Instruction Address 25
intermediate blocks 85
J
Java variable types 41
Java Virtual Machine 1
Javadoc 80
JBA 27
JBB 27
JBC 27
JBD 27
JBR 24
JBUM 9
job 26
Job Buffer A 27
Job Buffer B 27
Job Buffer C 27
Job Buffer D 27
Job Buffer Memory 9, 10
Job Buffer R 24
Job buffers 32, 52
Job handling 26, 52
Job table 27, 32, 52
Job thread 75
JobBuffer 80
JobHandler 79
JobTable 80
JVM 1
L
last block 85
live object 59
Local Signal Number 25
LSN 25
M
Main 79
Main memory 9
Mark-compact garbage collection 97
Mark-sweep garbage collection 97
multiple signal 22
O
object pool 51
P
pause time 96
phase division 87
PLEX 13
Plex 13
pointer 20, 49
preemptive 62
primitive type 41
priority queue 55
program sector 14, 38
Program Store 9, 10
Programming Language for EXchanges 13
PS 9
Index

R
R variable 15
Record 20.49
Reference Store 9.10
reference type 41
reflection 36
Regional processor 7.10, 77
Register Memory 9
relative time queue 33
RELOAD 18, 47
restart 17
RM 9
RP 7
RP signals 24
RP thread 74
RP-CP signals 24
RS 9
S
SDT 25
Signal 21.49, 80
Signal Addressing 25
Signal data 21
signal description document 14
Signal Distribution Table 25
Signal properties 21, 58
Signal Sending Pointer 26
Signal Sending Table 26
signal survey document 14
Signal types 24
single signal 23
Singleton 37
source code 81
source code document 14
SSP 26
SST 26
stack variable 45
start 17
starvation 62
STATIC 19, 48
String variable 16, 44
subfunction 27
subprogram 14
subroutine 46, 67
Symbol variable 15, 43
Synchronization 68
T
TEMPORARY 17, 45
THL 27
Threads 59
throughput 96
Time Queue A 33
Time Queue B 33
Time Queue C 33
Time Queue D 33
Time queues 33, 56
TimeQueueAbsolute 80
TimeQueueRelative 80
TQA 33
TQB 33
TQC 33
TQD 33
Traffic-handling level 27
TRANSIENT 18, 48
U
Unicode 44
unique signal 22
Usage 77
V
Variable 14
Variable addressing 19
Variable properties 16, 44
Variable type 15, 41
Index

Run-time environment for Plex-C on JVM
Sammanfattning

Abstract

The Ericsson AXE-based systems are programmed using an internally developed language called Plex-C. Plex-C is normally compiled to execute on an Ericsson internal processor architecture. A transition to standard processors is currently in progress. This makes it interesting to examine if Plex-C can be compiled to execute on the JVM, which would make it processor independent.

The purpose of the thesis is to examine if parts of the run-time environment of Plex-C can be translated to Java and if this can be done so that sufficient performance is obtained. It includes how language constructions in Plex-C can be translated to Java.

The thesis describes how a limited part of the Plex-C run-time environment is implemented in Java. Optimizations are an important part of the implementation. It is also described how the JVM system was tested with a benchmark test.

The test results indicate that the implemented system is a few times faster than the Ericsson internal processor architecture. But this performance is still not sufficient for the JVM system to be an interesting replacement for the currently used processor architecture. It might still be useful as a processor independent test platform.

Nyckelord

Keyword

AXE, APZ, APZ VM, Plex-C, processor architecture, run-time environment, compiler, JVM, Java, threads, garbage collector, data structures, algorithms, optimization, Ericsson, SoftLab.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Johan Möller