

MASTER'S THESIS

Software Development for Embedded Systems Based on ECMA 335

Stefan Richter

September 29, 2006

Supervisors: Prof. Dr. rer. nat. habil. Andreas Polze
Dipl.-Inf. Andreas Rasche

Abstract

While a lot of research has been done on making programming of standard computer systems simpler, less error-prone, and more efficient, support for embedded systems development is still very poor. This is because that discipline's special needs, such as direct hardware access, are rarely considered in theory and practice. In particular, virtual machines usually do not allow for accessing hardware directly, making it impossible to express substantial parts of embedded systems *inside* the virtual environment. By specifying additional rules, describing an implementation of a conforming compiler, and presenting examples, we show how the virtual machine defined by the ECMA standard 335 can be carefully extended to support hardware-near programming.

Zusammenfassung¹

Während sehr viel Forschung betrieben wurde, um die Programmierung von Standard-Computersystemen einfacher, weniger fehleranfällig und effizienter zu gestalten, lässt die Unterstützung für die Entwicklung eingebetteter Systeme noch immer zu wünschen übrig. Der Grund dafür ist die Vernachlässigung der besonderen Bedürfnisse dieser Disziplin in Theorie und Praxis, wie beispielsweise der direkte Zugriff auf die Hardware. Im besonderen erlauben es virtuelle Maschinen für gewöhnlich nicht, Hardware direkt anzusprechen, und machen es somit unmöglich, wesentliche Teile eingebetteter Software *innerhalb* der virtuellen Umgebung auszudrücken. Anhand zusätzlicher Regeln, der Beschreibung eines entsprechenden Compilers und einiger Beispiele zeigt der Autor, wie die durch den ECMA-Standard 335 definierte virtuelle Maschine sorgfältig erweitert werden kann, um hardwarenahe Programmierung zu unterstützen.

¹German translation of the abstract as required in the examination regulations.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Embedded Systems	3
1.3	State of the Art	5
1.4	General Notes	6
1.5	Organisation	7
2	Related Work	8
2.1	Embedded Systems and Java	9
2.2	Embedded Systems and ECMA 335	11
3	Environment	14
3.1	ECMA 335	15
3.2	GNU Compiler Collection	25
4	Extensions to ECMA 335	27
4.1	Design Goals	28
4.2	Hardware Access	30
4.3	Interrupt Handling	39
4.4	Concurrency	43
4.5	Summary	47
5	Implementation	48
5.1	Front End Overview	49
5.2	Extensions	54
5.3	Status	55
5.4	Performance Evaluation	56
6	Examples	65
6.1	Hardware Access	66
6.2	Interrupt Handling	71
6.3	Scheduling	75

7 Conclusion	79
7.1 Summary	80
7.2 Outlook	80
A Classes for the Renesas H8/3297 series	81
Bibliography	89

List of Figures

1.1	Embedded systems	4
1.2	Internal view of an embedded system	5
3.1	Relations between CTS, CLS, and high-level languages	16
3.2	Types in the CLI	18
4.1	Components as a tree	34
4.2	Stacks when switching the context	44
4.3	Contexts on the stack	45
5.1	Structure of the front end	51
5.2	Frequency distribution of execution times of test runs	58
5.3	Graphical display of compiler execution times (test series 1)	60
5.4	Graphical display of compiler execution times (test series 2)	62
5.5	Graphical display of compiler execution times (test series 3)	63

List of Tables

3.1	CLI's built-in types	17
5.1	Compiler execution times (test series 1)	60
5.2	Compiler execution times (test series 2)	61
5.3	Compiler execution times (test series 3)	64
6.1	Frequency Ranges of the RCX's 8-bit Timer	67
6.2	Cycles in every path	71

Chapter 1

Introduction

1.1 Motivation

For various reasons, software for embedded systems seems to be more difficult to develop than “traditional” desktop systems. WOLF lists many of them [WOLF, 2001, p 5, 9], eg:

- Restricted development environments: Code must often be crosscompiled on a proper workstation, then downloaded to the embedded system, which reduces development speed substantially.
- Limited observability and controllability: Often without proper I/O devices, embedded systems are hard to observe for debugging and testing purposes.
- Complex testing: Embedded systems must be tested in their environment, timing is often important.
- Real time: Many tasks must be completed before a certain deadline.
- Complex algorithms: Physical environments often require sophisticated control algorithms.
- Manufacturing costs: Embedded systems are often mass-market products, the overall cost for the system must be minimised.
- Power: Battery life, heat consumption, and general cost (eg for a larger power supply) are all affected by the power consumption.

This is further complicated by a lack of high-level language support for crucial aspects of embedded systems such as direct hardware access. The major goal of this thesis is to make the power of high-level languages such as C# [ECMA 334] available for embedded systems by proposing extensions to the Common Language Infrastructure (CLI) described in ECMA standard 335 [ECMA 335]. Besides their implementation, we give proof-of-concept examples of embedded software developed using these extensions.

As this task in its full complexity would be quite large an endeavour for the extent of a Master’s thesis we shall concentrate on those topics that are the most basic ones for the development of embedded systems: hardware access, interrupt handling, and preliminaries for concurrency.

A major part of the software developed in the course of this work were intended to be a contribution to a much larger project: The OPERATING SYSTEMS AND MIDDLEWARE GROUP at HASSO PLATTNER INSTITUTE [OSMG, 2006] is currently developing a front end (LEGO.NET) for the GNU COMPILER COLLECTION (GCC) [GCC a, 2006]. Once finished, this front end will be able to parse binary files of the CLI and, along with the GCC back ends, to generate binary executables for any platform supported by the GCC [GCC b, 2006]. These executables are supposed to comply with MONO’s implementation of the CLI [Mono, 2006].

However, during the process of getting familiar with the GCC and the LEGO.NET front end, a new front end evolved out of the tests we conducted to understand the basics of the GCC concerning front end development. At a certain point, it seemed easier to implement missing functionality in our front end instead of extending the other front end. Additionally, we were not quite happy with the other front end's structure and approach to certain aspects. That is why we finally implemented our own front end.

1.2 Embedded Systems

Originally, computers were invented for helping with large and tedious calculations; given a certain input, their purpose was to present the right result. As LEE points out, most efforts in computer science have been spent on these *results-oriented* computations: theoretical and formal models, programming languages, and software engineering methodologies [LEE, 2002]. Later on, with computers becoming more powerful, *process-oriented* applications, such as (graphical) user interfaces or computer games, increased in number. The simultaneous miniaturisation of processors made their use sensible to overcome drawbacks of standard approaches in electrical engineering. In particular, their programmability allows for faster development and cheaper construction of controller logic.

Since such systems, consisting of one or more processors, other circuits and software, are embedded in a physical environment that they are supposed to control they are called *embedded systems*. The same reason is the root of all evil that makes embedded systems development a difficult task. It becomes manifest in two issues that application programmers for servers and desktop computers usually need not care about: real-time issues and heterogeneous hardware that must be accessed, more or less directly.

In contrast to results-oriented or other process-oriented systems, time is an important entity in real-time systems; not only must a computation render a correct result or behaviour but it must do so in a limited period of time, ie there is a deadline until when a result must be readily computed or a process be completed. Because real-world processes are rarely linear or sequential, real-time often involves concurrent execution of different programs; while the system acts in response to an event of the environment another event might happen, which might need preferential treatment. Fortunately, we will not further discuss the complex issues of real-time systems as this thesis concentrates on the other aspect of embedded systems.¹

In the following, we consider an *embedded system* a system that has been designed

¹In this thesis, we frequently refer to real-time systems although its subject is *embedded systems*. This is because embedded systems are quite often real-time systems, and literature often treats embedded systems' issues on the fringes of real-time systems. Therefore, especially when conferring to related work, we need to talk about real-time systems when we would rather like to look at embedded systems neglecting real-time aspects.

to control a part of the physical world.² While some systems might be controlled by user input (eg the logic of a climate control system) others are built to perform their tasks autonomously (eg certain control systems in cars or airplanes). Figure 1.1 shows an embedded system.

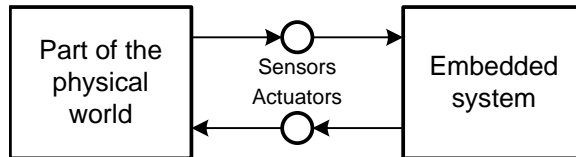


Figure 1.1: Embedded systems

As you can see, we have the *environment* and the *embedded system*. To keep terms simple we call each device that can change the environment an *actuator*. On the other hand, a *sensor* is a device the state of which is changed by the environment.³ With its sensors, the embedded system can collect information about the environment such that it can react on events using its actuators. In general, this can either be done by checking the sensors in intervals (*polling*) or by waiting until a sensor signals a change (*interrupt driven* or *event driven*).

From the internal point of view, an embedded system can be partitioned into three major parts: the CPU, the actuator logic, and the sensor logic (see figure 1.2). For our purposes, the CPU shall comprise any chip that is involved in *processing* the control algorithms, such as an FPU in INTEL's i386 series, while actuator logic and sensor logic are those parts of the embedded system that are in autonomous control of actuators or sensors, respectively. They are connected to the internal data bus (or possibly buses) the CPU writes to and reads from.

For practicability reasons, we only want to consider embedded systems that contain a CPU that operates on 8-bit or larger words because the smallest piece of data we can handle in the CLI is 8 bit in size.

²HSIEH, BALARIN & SANGIOVANNI-VINCENTELLI say: "Embedded systems are loosely defined as any system that utilizes electronics but is not perceived or used as a general purpose computer." [HSIEH, 2001] We very much dislike such a negative definition. Moreover, there are (not totally pathological) counterexamples: a pocket calculator is certainly not a general purpose computer and will hardly be considered an embedded system.

³We can also consider a timer module *in* the embedded system a sensor since the physical processes in it are in a certain sense part of the environment. In a more rigorous way, even I/O components could be considered sensors/actuators. Then, every computer is an embedded system. Perhaps this point of view could actually lead to better device drivers.

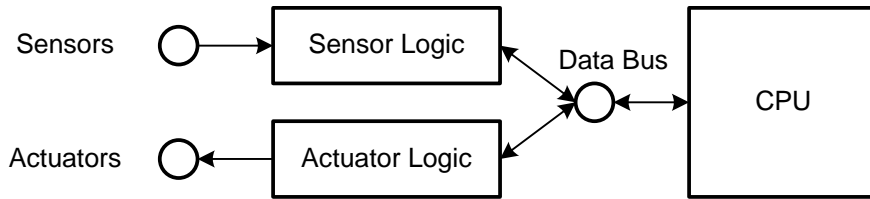


Figure 1.2: Internal view of an embedded system

1.3 State of the Art

As many sources support, the dominant programming language for embedded systems is C with efforts moving towards C++ (eg [NILSEN, 2005][Komodo, 2006]). Introductory literature can be found in abundance regarding that topic and is still produced (eg [BARR, 2006] or [BOLLOWS, 2006]), even though academic support and research is very low. C's main advantage is its flexibility in comparison to other high-level languages. Due to this flexibility, it is quite easy to interoperate with other languages and hardware. However, this advantage often easily turns into a disadvantage; because there are so few restrictions, there are almost no means to prevent the programmer to make mistakes. Wild pointers and erroneous memory management are common sources for bugs that are very hard to find. Moreover, C offers little support for modularisation making projects more complex than necessary.⁴

C++ introduced a lot of enhancements to C, most of which support modularisation and reusability of code. Still, many sources of errors could not be eliminated, because one major goal of C++ was a high degree of compatibility to C and existing code in that language [STROUSTRUP, 2000]. Nevertheless, C/C++ are often a good choice for implementing short pieces of code that cannot be expressed in safer languages such as JAVA. For instance, the JAVA NATIVE INTERFACE (JNI) provides for a way to extend the JAVA VIRTUAL MACHINE (JVM) by arbitrary modules; mostly, C/C++ are used for that purpose.

Talking about JAVA, it should be noted that this easy-to-learn language and execution system is highly favoured by some parts of industry and academia. Its automatic memory management and restrictive virtual execution environment solve quite a few important problems of C/C++. Unfortunately, these mechanisms inhibit real-time and embedded systems development; but research for making JAVA suitable for real-time systems is under way (eg [RTJS, 2000][JVMRTS, 2001][BACON, 2005]).

⁴A lot of problems that arise in C and C++ can be avoided if the programmers work very disciplined. Unfortunately, there is no way to make them do so. In fact, that is one of the jobs of language design and compilers. Languages can never add functionality to what we can do in low-level assembly languages. All they can do is to provide for more effective means of expression and to disallow certain instruction sequences.

Another important but more academic programming language for embedded systems is ADA. In contrast to JAVA, this language provides for mechanisms to directly access hardware and define interrupt handlers. It further contains a unique approach to concurrency and synchronisation issues. However, ADA seems not to be widely accepted as an industrial tool—it is arguably too complex—but has a strong academic support, though. Interestingly, comparisons between ADA and JAVA are rather pro-ADA [PINHO, 1999][POTRATZ, 2003].

We do not want to conceal that there are lots of other programming languages and/or systems for developing embedded software. However, since they are rarely used by a greater community, documentation is often spare and hard to access.

Further it should be noted, that there is much more to embedded software development than programming. For instance, this includes requirements engineering, software management, and system testing. Nevertheless, while these processes, such as CMMI, have high significance for embedded software that must provide for certain guarantees in terms of reliability, safety, and other issues, they are common to other software as well. Therefore and since they are not important to the subject of this thesis we will not discuss them in greater detail.

More specific to embedded software is the process of hardware/software co-design. As the name suggests, hardware and software are developed at the same time, more or less dependent on each other[WOLF, 2001][LAVAGNO, 2005]. With that, optimised embedded systems can be produced, which fit much better in their environment without sacrificing flexibility. Nevertheless, the choice of programming languages is seldom restricted, and hence as above.

Often, such processes are accompanied by formal and semi-formal methods, for instance, the Specification and Description Language (SDL), the Unified Modeling Language (UML), or mathematical models. For example, there are attempts to use temporal logic in the specification process of embedded systems [BELLINI, 2000].

1.4 General Notes

We consider a program to be a description for activities a CPU shall perform. Hence, the term *hardware* will never include the CPU that is running the respective program but all other parts the CPU is performing a part of its activities on. This allows a clear separation of concepts.

Throughout this document, we will adopt the C# convention and use the short names of attributes⁵, ie `MemoryAlias` instead of `MemoryAliasAttribute`. Moreover, for ease of reading, we will not always give entities' full names, in particular if the namespace can easily be deducted from the context or the entity itself.

⁵see section 3.1.3.3

In an ideal course of action in embedded software development, we favour a strong division of labour: Every piece of hardware shall be completely specified by a CLI library produced by the manufacturer or any other person feeling appointed to that job. With that, embedded systems programmers need not know about locations of hardware registers or special access to them. Instead they can program embedded systems as they would do with ordinary systems, by using identifiers. As we will see later, this is possible and sensible.

1.5 Organisation

Chapter 2 presents a short overview on some related work in the field of machine-oriented programming in the context of virtual machines. In chapter 3, we describe the environment our work is placed in, ie the ECMA standard 335 and the GNU Compiler Collection. We then motivate extensions for the ECMA 335 in chapter 4 along with giving precise and formal descriptions of them. In chapter 5, we dig into the implementation of our front end and explain important aspects. Some practical applications of the extensions to the LEGO RCX are given in chapter 6, whose translations are thoroughly analysed and evaluated. Finally, we give an overall performance evaluation, a summary and an outlook in chapter 7.

In short, the most important and challenging part of this thesis is chapter 4. Purpose of chapters 2 and 3 is to prepare readers for chapter 4, giving them insight about the necessity of our extensions and introducing them to the ECMA standard 335 as so much as we feel appropriate in order to understand the *essence* of the extensions. On the other hand, chapters 5 and 6 are intended to justify that the results presented in chapter 4 are indeed sensible, by showing that a compiler providing support for them is efficiently implementable without producing exorbitant overhead in the generated translation.

Chapter 2

Related Work

2.1 Embedded Systems and Java

Due to the commonalities of both virtual machines, SUN's JAVA and the CLI, and because of the much greater popularity and more widespread use of JAVA, we first give an overview on the work that has been done in order to use JAVA in embedded and, more often, real-time systems.

2.1.1 The NIST Report

At the end of the last century, the NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY (NIST) sponsored the REQUIREMENTS WORKING GROUP FOR REAL-TIME EXTENSIONS FOR THE JAVA PLATFORM involving representatives of many of the most influential companies in that field as well as academic institutions. This joint effort resulted in a report specifying requirements for real-time extensions for JAVA [NIST 1999]. They explicitly do not specify a standard for those extensions but restrict themselves to requirements engineering.

Besides giving a comprehensive set of definitions for terms and concepts commonly used in real-time computing, this report clearly states its guiding principles, which might be considered a sound base for any framework:

1. The design of RTJ¹ may involve compromises that improve ease of use at the cost of less than optimal efficiency or performance.
2. RTJ should support the creation of software with useful lifetimes that span multiple decades, maybe even centuries.
3. RTJ requirements are intended both to be pragmatic, by taking into account current real-time practice, and visionary, by providing a roadmap and direction to advance the state of the art.

“Because of the wide variety of applications that need to be served by these requirements”, the report proposes that a standard for real-time extensions for JAVA be divided into a minimal set of profiles for specific needs and a common core that all profiles share. Interestingly, these profiles may not only extend the core's functionality but also restrict it. Each conforming implementation must implement the core and one or more profiles, examples of which are high availability, low latency, or distributed real-time.

In December 1999, PINHO & VASQUES published a comparison about the suitability for real-time programming of JAVA and ADA [PINHO, 1999]. Being seemingly ADA evangelists, their conclusion sees major advantages of that language over JAVA when it comes to software *engineering*. Since then, some of their objections have been added to newer versions of JAVA, eg generics, and others can be found in the CLI.

¹The footnote reads: “As defined in the Terms section, *RTJ* is used to identify the sum of the functionality and services that would be provided through real-time extensions for the JAVA platform.”

2.1.2 The Real-Time Java Specification

Based on the NIST report, the Real-Time Java Specification proposed enhancements for scheduling threads, memory management, asynchronous computation, and direct memory access, suitable for real-time systems [RTJS, 2000]. In general, these enhancements lay on a higher level of abstraction than our extensions; for example, there are no provisions for direct access to port-based I/O, interrupt handling or the implementation of a scheduler other than using the `JAVA NATIVE INTERFACE` (JNI). The JNI is JAVA's way of making the virtual machine interact with other, native software [JNI, 2003].

The means of choice for low-level access to hardware is a class `RawMemoryAccess`, that has methods for writing to or reading from certain hardware locations specified by their offset. There is a pair of getters and setters for 8-, 16-, 32- and 64-bit sized locations as well as arrays of those. The specification reads that “[t]he `RawMemoryAccess` class allows a real-time program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software.” Still, port-based I/O is to be implemented by JNI methods.

After having compared the RTJ specification to features of ADA, BROSGOL & DOBBING observe an evolution of JAVA and ADA towards each other [BROSGOL, 2001]. They write that “JADA's” future lies in the co-operation of both languages. We may add that language interoperability is one of the major goals of the CLI.

There are quite a few implementations compliant with the specification. First of all, the official reference implementation was developed as a bytecode interpreter by TIMESYS [TimeSys, 2006]. Based on SUN's `JAVA MICRO EDITION`—a limited version of the JAVA virtual machine—and running on the LINUX operating system, it comprises all the features mandatory for compliance with the specification.

At the University of California, Irvine, an extension for the GNU Compiler for JAVA has been developed [CORSARO, 2002]. Similar to the approach of our front end, JRATE is an ahead-of-time compiler producing fully compiled and linked images that can be put on the target platform. In addition, CORSARO & SCHMIDT extensively display results obtained from the performance tests they conducted with their open-source tool RTJPERF on JRATE, the reference implementation, and the `C VIRTUAL MACHINE`, a non-conforming real-time JAVA VM [CORSARO, 2002a].

SUN also published `JAVA SE REAL-TIME`, arguably “the first conformant commercial implementation of Java Specification Request (JSR) -001, the Real-Time Specification for Java (RTSJ)”, which is only available for machines running the SOLARIS 10 operating system [JSERT, 2006]. It cannot be considered for small embedded systems since it has a footprint of several megabytes.

German company AICAS have implemented the RTJS in the JAMAICA VM. They write that “[t]he JamaicaVM is the only implementation that provides hard realtime guarantees for all features of the languages together with high performance runtime efficiency.” [AICAS a, 2006]. They provide for a linker that can include JAVA bytecode as an array

in the VM executable in order to get a single binary for VM and application. However, with its memory footprint of about 128 kB it is not suited for smaller embedded systems, such as the LEGO RCX [AICAS b, 2006].

AJILE is a microprocessor architecture executing JAVA bytecodes directly. It is based on the RTSJ and aimed at real-time and network applications[HARDIN, 2005]. CHIAO ET AL describe their implementation of the RTSJ [CHIAO, 2002].

2.1.3 Other Approaches

JX is an operating system based on the JAVA Virtual Machine. It was developed at the University of Erlangen, Germany, emerging in 1999 from the METAXA VM built in 1996 [JXhist, 2006]. There has been no official release, yet, but version 0.1.1 has been published under the GNU General Public License. Since there are newer papers (*Studienarbeiten*) from 2005, we assume there is still work going on. The JX operating system consists of a microkernel and several *domains* [GOLM, 2002]. Instead of using JAVA's byte arrays for representing memory, it uses custom classes, operations of which are treated specially by the compiler. For example, there is a class for memory mapped I/O; the compiler will not re-order or omit any operations of that class. Port-based I/O is accessed by microkernel operations that the compiler can inline. Interrupt handling and schedulers can be implemented in JAVA using special methods.

Australian-based company RTJ COMPUTING offers SIMPLERTJ, a reduced implementation of the virtual machine designed for 8 or 16 bit embedded systems with limited memory [RTJCom, 2006]. It interprets the JAVA bytecode, but uses prelinked classes, such that the start-up time on the target system is reduced, because of the assumption that embedded systems rarely require *updates* but more often *restarts*, eg in case of turn on/off. Hardware access is realised by a technique similar to JNI.

Real-time behaviour is a main concern of IBM's METRONOME garbage collector [BACON, 2005]. Other virtual machines for JAVA in embedded systems include AONIX PERC [NILSEN, 2005], ESMERTEC JBED [Jbed, 2006], and SUN JAVA MICRO EDITION[JME, 2006]. The KOMODO project is an effort to build microcontrollers running JAVA applications directly [Komodo, 2006].

2.2 Embedded Systems and ECMA 335

Close to the matter of this thesis, LUTZ & LAPLANTE investigate how well the programming language C# and MICROSOFT's implementation of ECMA 335, .NET, are suited to the needs of real-time systems development [LUTZ, 2003]. They conclude that a lot work has to be done in order to make those a helpful tool for real-time systems development. Hence, they should not be used in hard real-time systems but for selected soft real-time systems only, provided the developers maintain a high programming discipline.

Essentially concentrating on schedulability and related topics such as deadlock and priority inversion prevention, they point out some critical issues: non-deterministic memory management, only five thread priorities, no support for runtime code analysis, no properly working thread inheritance, insufficient timer accuracy. On the other hand, they note some advantages the CLI could bring to real-time systems development, most notably exception handling and type safety. They do not cover need and support for basic I/O operations.

In 2005, ZERZELIDIS & WELLINGS published an adaptation and extension of the NIST report for the CLI [ZERZELIDIS, 2005]. They concentrate on real-time requirements, observing that there already is a similar structure to the NIST report's profiles in the CLI. They enhance the NIST report's requirements by some important requirements such as the CLI's feature of CLS compliance for libraries. Nevertheless, they do not specify any requirements for accessing hardware directly.

VON LÖWIS & RASCHE present their LEGO.NET project and the CLI front end for the GCC that they have implemented; for various constructs, they analyse in greater detail the complexity of the execution times [ISORC, 2006]. There is no support for I/O operations other than access to external functions—one of the major reasons for this thesis.

The .NET COMPACT FRAMEWORK is MICROSOFT's implementation of the CLI for smaller-than-PC-size embedded systems or resource-limited PCs running the WINDOWS CE operating system. It has some more functionality, such as access to infrared devices, but in general, it is a less comprehensive version of the .NET FRAMEWORK [.NETCF, 2006]. In particular, there are no extra provisions for accessing hardware other than the platform-invoke mechanisms.

In 2005, MICROSOFT RESEARCH presented an operating system 95% of which were written in C# with the remaining code being C++ and assembly code.

In a short email conversation, Galen HUNT, one of the two project leads, writes about the more hardware related parts:

We have assembly code in the same places that any operating system does: the thread context switch code, the bottom edge of the interrupt vector, and to issue instructions that can't be expressed in C# or C++ (for example the "in", "out", "sti", and "cli" instructions and the instructions to read and write cr0, cr2, cr3, etc.). We use C++ code to initialize the context for C# code to run. We also use C++ code for math library code that control the FPU as these instructions that have no MSIL equivalent.

He also gives a C# example to further illustrate this.

```
namespace Microsoft.Singularity
{
    public class Processor
    {
        public static extern bool DisableInterrupts();
    }
}

namespace Microsoft.Singularity.Io
{
    public sealed class IoPort
    {
        public void Write32(uint offset, uint value)
        {
            if (!writable || offset + 4 > size)
            {
                Error.WriteNotSupported();
            }

            HalWriteInt32(port + offset, value);
        }

        private static extern void
        HalWriteInt32(uint port, uint value);
    }
}
```

The external methods above are implemented using C++ and assembly code:

```
bool
Class_Microsoft_Singularity_Processor::g_DisableInterrupts()
{
    uint32 eflags;
    __asm
    {
        pushfd;
        pop eax;
        mov eflags, eax;
        cli;
    }
    return (eflags
        & Struct_Microsoft_Singularity_X86_EFlags_IF) != 0;
}

void
Class_Microsoft_Singularity_Io_IoPort::
g_HalWriteInt8(uint32 port, uint8 value)
{
    __asm
    {
        mov edx, port;
        mov al, value;
        out dx, al;
    }
}
```

Chapter 3

Environment

This chapter gives an overview on the standard ECMA 335 and—much shorter—the GNU Compiler Collection. In order to prepare the ground for the extensions presented in chapter 4, we need to make clear certain details of the standard that are important for understanding why the extensions evolved in the fashion they did. Therefore, we recommend the reading of that chapter even to readers who already know about the standard unless they have a very strong background including detailed knowledge of the execution engine.

3.1 ECMA 335

The ECMA standard 335 describes the Common Language Infrastructure (CLI), which specifies an execution environment—the Virtual Execution System (VES)—along with the operations it can perform and the formats of the files it can execute. For this purpose, the description also encompasses a specification of the Common Intermediate Language (CIL) which contains the set of instructions that can be executed.

The standard consists of six partitions:

- I. *Concepts and Architecture*. This partition describes the overall structure of the standard and the main concepts.
- II. *Metadata Definitions and Semantics*. The partition specifies completely the file format used for describing programs that a compliant virtual machine can execute.
- III. *CIL Instruction Set*. In this partition, you can find the operational semantics of every instruction that can be performed by a compliant virtual machine.
- IV. *Profiles and Libraries*. This partition consists of a very short textual part and a much longer XML file. These two documents together describe which profiles exist and what libraries a virtual machine has to provide for in order to be a conforming implementation.
- V. *Debug Interchange Format*. This very short partition defines the format that is used for transmitting debug data between CLI producers and consumers.
- VI. *Annexes*. Some examples and further information about certain aspects have been collected in that last partition.

In the course of this thesis, partition I and III are the most important ones. The version presented here is the third edition. In the following, we will concentrate on the technical aspects of the basic concepts which are most important when dealing with I/O, and that are necessary to understand the concepts developed in this thesis. Therefore, we will not cover most of higher-level issues, such as security mechanisms, application domains, generics, and other advanced programming paradigms.

There are compilers for a wide variety of languages to the CLI, most notably ADA, C, C++, C#, COBOL, FORTRAN, HASKELL, JAVA, JAVASCRIPT, LISP, MODULA-2, PASCAL, PHP, PROLOG, PYTHON, RUBY, SMALLTALK, and TCL/Tk [RITCHIE, 2006].

3.1.1 Type System

Because the CLI is intended to support different kinds of programming languages, it provides for a rich type system, called the Common Type System (CTS). On the other hand, the CLI is supposed to enable language interoperability, hence it specifies a set of rules, the Common Language Specification (CLS), that restrict the use of the CTS to a minimal subset that every language must support (see figure 3.1). Therefore, while any language may let its users access any means of the CTS it must make sure that parts intended for interoperability, ie those that can be used by other languages, are CLS compliant.

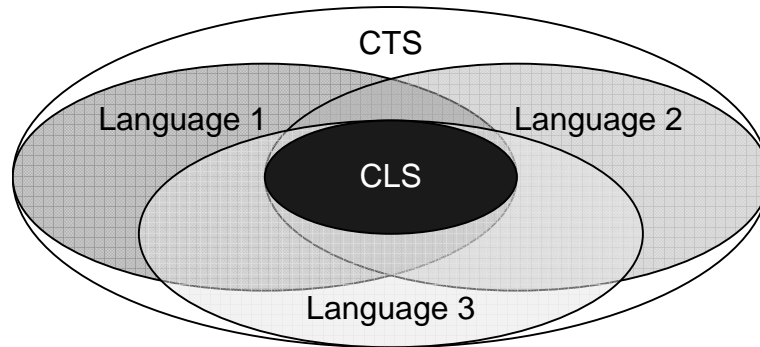


Figure 3.1: Relations between CTS, CLS, and high-level languages

Since some common terms are used quite differently in computer science, we feel obliged to display their meaning in the course of ECMA 335 whilst explaining the concepts.

3.1.1.1 Values

Most fundamentally, the standard uses the terms *value* and *location*. In contrast to its purely abstract meaning as in mathematics, a value in ECMA 335 is a logical representation of that mathematical value as a sequence of bits. For example, the number *two thousand and six* can be represented as 2006, MVI, or 0000011111010110. The last one would be considered a value in the CLI. However, there is no restriction on how this value is physically represented, avoiding the portability inhibiting commitment to an endianness. The VES is free to decide how it stores values in locations, and how it

reads or writes them. Examples for locations are local variables, method arguments, or evaluation stack slots, but also locations on the heap.

Values can be structured, ie their bit sequence might be subdivided into smaller values except for those of basic built-in types. Depending on whether these sub-values are named or indexed by an integer they are called *fields* or *array elements*. Accordingly, types that contain array elements are called *array types*; they cannot define additional named fields. Types that contain fields are called *compound types*. Fields and array elements have never-changing types, and for all elements of an array type that type is the same.

3.1.1.2 Types

A *type* describes a set of values, which are said to be *instances* of that type, along with the operations that can be performed on them. Each value can have more than one type, but it has one *exact type* that completely specifies the set of operations that can be performed on that value. Whenever a value is used, be it as an operand or as an argument, one of the value's type, but not necessarily its exact type, must be known. The VES is therefore strongly typed. Table 3.1 lists all types that are built in to the CLI (their full library names have the prefix `[mscorlib]System.`). As you will notice, not all of them are CLS compliant.

Name	Library Name	CLS	Description
<code>int8</code>	<code>SByte</code>	-	8-bit signed integer
<code>int16</code>	<code>Int16</code>	+	16-bit signed integer
<code>int32</code>	<code>Int32</code>	+	32-bit signed integer
<code>int64</code>	<code>Int64</code>	+	64-bit signed integer
<code>uint8</code>	<code>Byte</code>	+	8-bit unsigned integer
<code>uint16</code>	<code>UInt16</code>	-	16-bit unsigned integer
<code>uint32</code>	<code>UInt32</code>	-	32-bit unsigned integer
<code>uint64</code>	<code>UInt64</code>	-	64-bit unsigned integer
<code>native int</code>	<code>IntPtr</code>	+	native size signed integer
<code>native unsigned int</code>	<code>UIntPtr</code>	-	native size unsigned integer
<code>bool</code>	<code>Boolean</code>	+	8-bit two-valued type
<code>char</code>	<code>Char</code>	+	16-bit Unicode
<code>float32</code>	<code>Single</code>	+	32-bit floating-point
<code>float64</code>	<code>Double</code>	+	64-bit floating-point
<code>object</code>	<code>Object</code>	+	object type root
<code>string</code>	<code>String</code>	+	Unicode strings
<code>typedref</code>	<code>TypedReference</code>	-	typed reference

Table 3.1: CLI's built-in types

ECMA 335 distinguishes between *value types* and *reference types*, the difference being that values of value types contain directly their data while a value of a reference type always describes a location of another value. This is, in a certain way, comparable to the datatype concepts in C where we have simple types and structs (value types) as well as pointer types (reference types). As in C, the most important implication is that values of value types need to be copied whenever they are passed as arguments while in case of reference types only a reference is passed. Nevertheless, the concept of reference types is more sophisticated since their use is quite restricted. In figure 3.2, you can see a partitioning of the CLI's type system; three dots indicate that there is more, albeit unspecified, than the listed subsets.

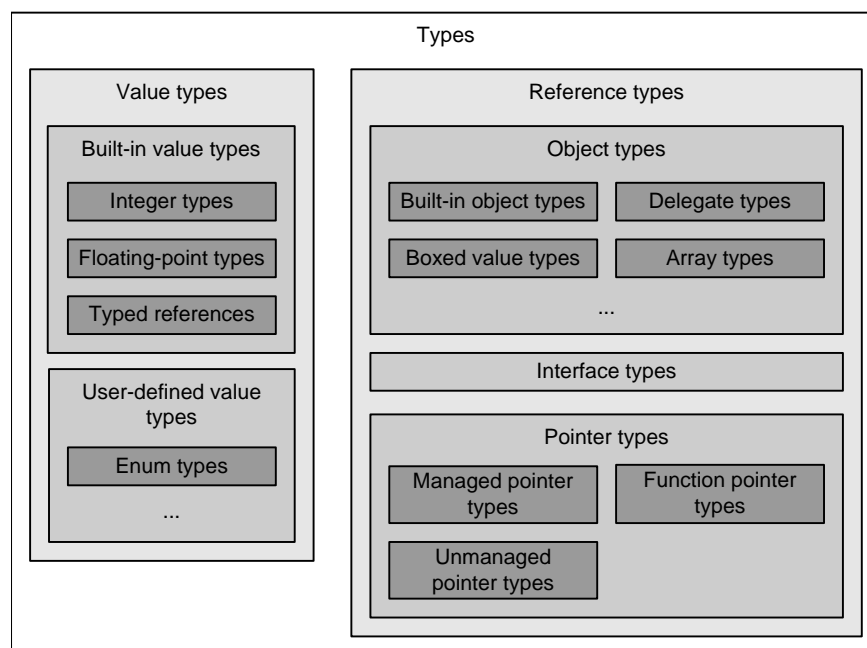


Figure 3.2: Types in the CLI

As displayed in that figure, there is a special kind of value types, the *enum types*. These types are essentially alternate names for existing types as each of them must contain precisely one field the type of which is the underlying type.

Of the reference types, *object types* are the most prominent representatives besides *interface types* and *pointer types*.¹ Instances of object types are self-describing, ie the VES can always determine the type of such an instance. Moreover, every object type has exactly one *base class* which it *inherits from* (*is derived from* or *extends*), except for

¹The standard further lists *built-in reference types*, such as `System.Object` and `System.String`. Since all of them are object types we took the liberty of including them in the object types group.

the class `System.Object`, which has no base class. Hence all object types inherit directly or indirectly from the `System.Object` class. If an object type A inherits from another type B it contains at least the same fields as B and all instances of A are also instances of B .

All instances of object types are created in a completely runtime-managed memory area, the *heap*. While the creation of objects is triggered by running programs, there is no way of explicitly removing an object from the heap. Instead, the runtime provides for an automatic memory management, that keeps track of references to an object. Once an object is not referenced anymore, it can be removed by the garbage collection.²

In general, it is possible to convert instances of value types into instances of object types (also called *objects*). The CLI provides for a mechanism called *boxing*. Along with each value type comes an implicit definition of an object type that is derived from `System.ValueType` (or, in case of enums, `System.Enum`, which extends that class). That object type contains a field the type of which is the value type. When boxing a value, a new object of the boxed type is created on the heap and the value is written in that field. Conversely, boxed values can be *unboxed*, returning a *managed pointer* to that field.

Managed and unmanaged pointers may refer to locations inside of objects. In contrast to unmanaged pointers, managed pointers are reported to the automatic memory management. Of course, neither managed nor unmanaged pointers are CLS compliant.

Although interface types describe values whose exact type is an object type, they are not object types themselves. Instead, they specify a contract that the object type has to fulfill when it *implements* an interface.

The third edition of ECMA 335 introduced *generics*. With these it is possible to make entities more abstract, allowing to instantiate them for specific types. Since this topic is beyond scope of this thesis we will not explain these sophisticated concepts in more detail. We also do not cover *reflection* mechanisms, which allows programs to discover types and other entities at runtime that have been unknown at compile time.

Lastly, we want to touch some predefined constants as they are referred to later: `null` is the value of object references that do not point to an object; `true` and `false` are the two possible values of `bools`. All basic numerical value types contain constant static fields `MinValue` and `MaxValue` that are set to the minimum and maximum value for that type. In addition, the floating-point types also define `PositiveInfinity` and `NegativeInfinity` for both infinities as well as `NaN` for values that are *not a number*.

3.1.1.3 Type Safety and Verification

In partition I § 6.1, the standard explains its view of type safety:

[...], from the point of view of the CTS, type safety guarantees that:

²This is a vast simplification. For example, the automatic memory management is allowed to remove objects that are not referenced but by each other.

- References are what they say they are.—Every reference is typed, the object or value referenced also has a type, and these types are assignment compatible (see 8.7).
- Identities are who they say they are.—There is no way to corrupt or spoof an object, and, by implication, a user or security domain. Access to an object is through accessible functions and fields. An object can still be designed in such a way that security is compromised. However, a local analysis of the class, its methods, and the things it uses, as opposed to a global analysis of all uses of a class, is sufficient to assess the vulnerabilities.
- Only appropriate operations can be invoked.—The reference type defines the accessible functions and fields. This includes limiting visibility based on where the reference is (e.g., protected fields only visible in derived classes). The CTS promotes type safety (e.g., everything is typed). Type safety can optionally be enforced. The hard problem is determining if an implementation conforms to a type-safe declaration. Since the declarations are carried along as metadata with the compiled form of the program, a compiler from the Common Intermediate Language (CIL) to native code (see 8.8) can type-check the implementations.

Furthermore, the standard clearly states its understanding of verification in partition I § 8.8:

Verification is a mechanical process of examining an implementation and asserting that it is type-safe. Verification is said to succeed if the process proves that an implementation is type-safe. Verification is said to fail if that process does not prove the type safety of an implementation. Verification is necessarily conservative: it can report failure for a type-safe implementation, but it never reports success for an implementation that is not type-safe. For example, most verification processes report implementations that do pointer-based arithmetic as failing verification, even if the implementation is, in fact, type-safe.

3.1.2 Methods

Methods specify the operations that can be performed on types; they contain a sequence of basic instructions each, that is executed sequentially using an *evaluation stack* for storing intermediate results. In addition, the virtual machine knows about *local* variables, *argument slots*, a *local memory pool*, and the heap that we mentioned above.

Instance methods have an implicit argument that provides for the value on which the operation is performed. The type of that argument is the type the method is defined on.

In case of *virtual methods*, the method itself is additionally determined depending on the exact type of that implicit argument; if a class *A* derives from a class *B* it may provide a new method definition for any of *B*'s virtual methods (*A overrides B*'s methods) and automatically inherits any virtual method it does not override. Note the difference to the JAVA platform, where all instance methods are virtual.

The evaluation stack operates only on six types: `int32`, `int64`, `native int &`, `0`, and `F`. `&` stands for managed pointers, `0` for object references, and `F` for a machine dependent floating-point type. Nevertheless, values of other types can be loaded on the stack as well; they are converted to the appropriate larger type. An implementation of the CLI is allowed to track these types in more detail, thus avoiding the penalties of using larger types for smaller ones.

Delegates are objects that contain a pointer to a method along with a reference to an object. They provide for runtime-defined methods for synchronously invoking the method, as well as starting, finishing, and aborting the invocation of the method. Thus, they constitute object-oriented and type-safe function pointers; while method pointers are not CLS compliant, delegates are. Whenever a method is called via a delegate the attached instance reference is passed as the implicit argument.

Besides any other mechanisms that are beyond the control of the virtual machine, there are two sources for possible interruption of a method's execution: *exceptions* and *thread switches*.

An exception is an object that CLS rules require to be of type `System.Exception`. Such an object is created by the program or the virtual machine and subsequently *thrown* to indicate to the caller that a problem has occurred. Further information about the problem shall be included by the thrower.

As soon as an exception is thrown, the execution of the instruction that is currently processed is aborted. If the instruction is contained in a *protected* block, the CLI discards the evaluation stack, but not locals and arguments, and searches for the first handler that has been (statically) registered with that block. Standard *catch* handlers handle the exception if the exception is of the given type including any subtypes; more flexible *filters* can include CIL code that checks whether it wants to handle the exception or not. If the handler does not handle the exception the next handler is searched for. In case there are no handlers that can handle the exception, the method is left and the exception is thrown with respect to the instruction that called the method, ie the same procedure is run for the calling method, and so on, until an appropriate handler is found. If no handler is found at all, the program is aborted with a calling-method stack trace. When a method's control flow is aborted and the method is left, *fault* handlers can be declared that execute code on exit of that method; *finally* handlers are executed whenever a method is left, regardless of usual or erroneous completion.

3.1.3 Modularisation

3.1.3.1 Modules & Assemblies

As modern systems tend to be large, the CLI supports refined means of modularisation. Every CLI file that contains type definitions is called a *module*. Nevertheless, a module is merely a physical entity, while logically, types are defined in *assemblies*. An assembly contains at least one module; a module can be contained in several assemblies. Current state is, however, that most assemblies are *single-module assemblies*, made up of precisely one module. Assemblies form the basic entities for language interoperability; CLS rules do not apply within an assembly, but only to those parts that can be accessed by other assemblies.

Every entity that is not a type must be contained in a type; and each type is either in a type (a *nested type*) or in an assembly. Still, every entity is referenced by its full name, which is composed of the assembly name, the type's name, and, if not a type, the entity's name. Namespaces, as common in some languages, do not exist but as part of type names.

Access to entities can be restricted as well. In general, `public` entities can be accessed by any other entity, `private` entities only by those contained in the same type, `family` ones by entities that derive from the type³. Additionally, there is an assembly-scope modifier as well as `and-` and `or-` combinations of it with `family` access.

Another means of modularisation is the use of global variables and methods. Such items are marked as `static` fields, methods, properties etc, stating that they are not part of an object. This implies that static methods do not have an implicit instance argument; delegates containing static methods ignore the reference to the instance. Like other entities, static entities must be part of a type, though. For static items that are not explicitly part of a type, there exists an implicit type `<Module>` in each module whose only purpose is to be a container for such items.

Assemblies come in two flavours: executables and libraries. The only difference is that executables must declare precisely one static method as entrypoint. There is no restriction on that method except on the signature; its return type is either `void`, `int32`, or `unsigned int32`, and it shall have no arguments or a vector of `strings` as argument. This allows for programs that integrate into common operating system environments.

3.1.3.2 Properties & Events

There are two minor, nevertheless extensively used, means of modularisation: *properties* and *events*. The main advantage of and reason for introducing properties and events is to enable support for nicer syntax in languages like C#.

³comparable to `protected` in C++, Java, and C#

Properties were intended to integrate getter and setter methods common in programming systems like `JAVA`; one method sets a private field, the other one returns its values. Both methods must have appropriate signatures, and their names shall follow a specific pattern. However, there are no additional restrictions to common methods on what a property's methods are allowed to do, so it is merely a structural means, especially when considering that properties need not define both, a setter *and* a getter method.

Akin to properties, an event consists of two methods for adding and removing delegates, that are called when the event's third method is called. Thus, events combine the methods necessary to model the publisher-subscriber pattern ([BUSCHMANN, 1996]).

3.1.3.3 Attributes

The standard fosters the massive use of *attributes*, which can be attached to any item, except attributes themselves:

Some languages predefine attribute types to represent language features not directly represented in the CTS. Users or other tools are welcome to define and use additional attribute types. [ECMA 335, partition II § 21]

While an attribute could be an instance of any type, CLS rules require an attribute to be an instance of an attribute type that inherits from `System.Attribute`. When attaching an attribute to an item, a constructor of the related attribute type is referred to along with an appropriate set of arguments. The meaning of code in that constructor is not generally specified; it is not quite clear whether an attribute *must* be present at runtime—accessible by reflection—or if they can be omitted at compile time, if the virtual machine feels to do so. Fortunately, we can leave this open since we do not bother with reflection.

3.1.4 Instruction Set

With respect to the type system, the instruction set has been divided into two parts: those instructions that can operate on entities of the numerical types and those that operate only on object types.

3.1.4.1 Basic Instructions

The first group contains the basic arithmetic operations addition, subtraction, multiplication, division, remainder, and negation. These operations are virtual in a certain sense, as their execution depends on the top element(s) of the stack; while integer and floating-point addition use the same instruction codes the operations performed are quite different. In case of integers, there are variants for overflow detection—throwing a `System.OverflowException` if the result cannot be represented in the result type— for

signed and unsigned integers⁴. Inconsistently, there are no division and remainder operations without overflow checks. For floating-point numbers, there is only one operation each, without overflow or divide-by-zero detection, as these cases are represented as infinities or NaN. Interestingly, the remainder operation for floating-point numbers does not follow the IEC 60559:1989 standard. Instead the method `System.Math.IEEERemainder` is to be used for that purpose.

Along with these arithmetic instructions, there are instructions for the basic bitwise operations conjunction, disjunction, exclusive disjunction, complement, left shift, and right shift with and without sign extension, that operate on integers only. There are also conversion operations, with or without overflow detection with respect to signed or unsigned integers. In particular, these operations allow narrowing of larger integers into smaller ones and conversion of floating-point numbers into integers.

Furthermore, there are operations that push values from locations on the evaluation stack or pop values from the evaluation stack into locations. These locations can be local variables, method argument slots, or indirectly referenced locations. In addition, there are instructions for pushing integer or floating-point constants, `null`, method pointers, or addresses of locations on the evaluation stack as well as for removing or duplicating its top element.

Moreover, this group contains comparison as well as conditional and unconditional branch instructions. While comparison instructions can be used to test for the state of being equal, lower than, greater than, the conditions of branches can further specify test for being not equal, not lower than, not greater than, true or not zero or not `null`, and false or zero or `null`. Except for the test for equality, all these instructions are available in a variant that performs the tests assuming unsigned integers or unordered⁵ floating-point numbers, respectively. As a speciality, there is an instruction that checks whether a floating-point number is finite, throwing an exception if it is infinite or NaN. Note that, for consistency reasons, targets of branch instructions are highly restricted. In essence, they may only point to the begin of each instruction in the same block, ie eg not out of their method, into or out of try, catch, finally blocks, etc. There is also a switch construct that can be used to implement jump tables.

In order to simplify life for CIL generators, ie high-level-language-to-CIL compilers, there are two instructions that allow for initialising and copying CLI allocated blocks of memory representing CLI structures. There is also an instruction to allocate memory local to the current method. The size of that memory is to be computed at runtime, and the memory is only valid during the runtime of the method. Thus, the instruction is an equivalent to C's `alloca` function, that allocates memory on the stack.

Finally, there are instructions to call methods directly or indirectly, leave methods

⁴Since the execution stack is not required to track whether integers are signed or unsigned there is need for a special operation for unsigned integers.

⁵This is only important if one or both operands are NaN. The first instruction would treat this as a negative outcome of the test whilst the unordered one would treat it as a positive one.

and special blocks in methods such as the try and catch blocks. Methods can also be jumped to, such that the caller discards its frame and passes it to the callee. Using that mechanism, some recursive methods can reduce linear spatial complexity to a constant one. There is also an instruction that allows a method to access its arguments in case that method has a variable argument list.

For the sake of completeness, we should mention the `nop` operation and the `break` instruction for use by a debugger.

3.1.4.2 Object-Model Instructions

Naturally, the type-system concepts are reflected in the instruction set. Most frequently used, there are load and store instructions for array elements, fields, and static fields, as well as general versions for any location the address of which has been pushed on the stack before. Note, that values of value types are always copied on the stack, even if they are greater than 64 bit. Further, there are instructions to load the addresses of array elements, fields, static fields, and string literals, the length of an array, and the runtime representation of any metadata token, such as methods, types, etc.

As described in section 3.1.2, a call to a virtual method involves the process of determining the object specific method. This has not been automatically integrated in the standard call instruction, in order to allow for direct invocation of virtual methods. Therefore, a special instruction has been introduced for that purpose. Another instruction performs half the work; it resolves the virtual method and loads a pointer to the respective method on the stack.

There are two instructions for creating objects and arrays, respectively. Objects can be checked for whether they are of a given type, and the size of each type can be determined at runtime. Furthermore, there are instructions for boxing and unboxing values.

Similar to the instructions that copy and initialise arbitrary blocks, there are instructions that copy instances of value types and initialise respective locations.

There are three instructions dealing with typed references: one constructs a typed reference out of type and address, and the other two get the address and the type out of a typed reference.

Finally, there are two instructions to throw/rethrow exceptions.

3.2 GNU Compiler Collection

The GNU Compiler Collection (GCC) is a set of tools that can translate programs written in several languages to several platforms. Common to all these tools is a core that performs the major translation and optimisation processes.

For every language supported, there is a *front end*, and for every target platform, there is a *back end*. Each front end's task is to read in the source file(s), check for syntax and other errors, and to build up an internal representation for the program to be further

processed by the core. In the end, the back end generates assembly code from the new internal representation the core produced. The kinds of representation used by front ends and back ends are fundamentally different.

At the moment, there are front ends for C, C++, OBJECTIVE-C, OBJECTIVE-C++, JAVA, FORTRAN, and ADA officially released in the main distribution [GCC c, 2006]. Further, there are front ends for PASCAL, MERCURY, COBOL, MODULA-2, VHDL, and PL/I. There are much more back ends, for instance for DEC ALPHA, ARM, RENESAS H8/300, INTEL I386, INTEL IA-64, MOTOROLA 68000, and SUN SPARC.

Chapter 4

Extensions to ECMA 335

Being the heart of this thesis, the following chapter introduces extensions for allowing access to hardware, programming interrupts, and the basics of concurrency. In general, we motivate each extension and explain in it informally before giving more rigorous and formal descriptions. More extensively commented and analysed examples can be found in chapter 6.

4.1 Design Goals

4.1.1 Real-Time Language Design Goals

Based upon HALANG & STOYENKO's concise requirements analysis for real-time languages [HALANG, 1991, pp 28–30], we propose the extensions with the following design goals in mind:

Reliability Due to the often critical tasks they are performing, real-time software must be very robust. Strong typing, structured system design, and error handling all support this goal.

Maintainability Usually, real time systems have a very long life span. Thus, their software should be easy to modify in order to fix errors or enhance their functionality. This applies especially to mass market products as these systems require constant improvements.

Modularity Modern, practically usable software is rarely developed by one person alone; even then, it tends to be large. Modularisation has turned out to be crucial for maintainability and organisation of work, eg for allowing partial, incremental compilation.

Simplicity Easy-to-understand concepts help programmers as well as compilers. Programmers can develop, maintain and modify much faster, while compilers can compile faster and produce better machine code.

Direct Hardware Access As they point out significantly: “Real time software typically interfaces many different hardware devices. It is important, therefore, to provide constructs in the language to access hardware locations and handle interrupts directly in a flexible and yet secure way.”

Predictability Along with functional properties, it is foremost important for real-time developers to guarantee temporal properties of their software. While we cannot forbid the implementation of non-analysable algorithms, we must provide for an adequately complete subset of both, existing and new, language constructs along with reasonable propositions about their temporal properties.

Concurrency In real-time systems, the execution of several tasks in a concurrent manner is an outmost important and widely (in newer works often implicitly) accepted paradigm [SHUMATE, 1992, p 256][STANKOVIC, 1998, sec 2.1][LIU, 2000][HANSSON, 2005].

Although we are mainly concerned with I/O of embedded systems, we still have to keep in mind that embedded systems are almost always real-time systems or have real-time issues to care about.

4.1.2 ECMA 335 Design Goals

Extensions to an existing framework should in general adopt the concepts, notions and paradigms of that framework and only invent new concepts whenever those are not sufficient for writing programs in an appropriate fashion. It is thus necessary to have a clear understanding of the “spirit” and the limitations of the CLI before attempting to enhance its expressiveness.

To summarise section 3.1, we consider the following paradigms as fundamental for the CLI:

Language Independence Features should be usable by a maximum of different types of programming languages and not inherently favour one over the others.

Platform Independence Programmers should in general not rely on features of the underlying platform.

Safety The execution environment must make sure that no program can affect others in a bad or serious manner. Program abortion for no matter what reason must be controlled.

Performance Execution of a program should accomplish results comparable to likewise safe and platform independent programs developed with similar efforts in high-performance languages.

Use of Metadata The heavy use of metadata improves all non-functional aspects of programs such as analysability and testing, and hence their safety.

It cannot be our intent to improve the CLI in those areas. Moreover, we will quite likely have to compromise platform independence and safety due to the nature of our task. However, we shall constantly check the figurative doors that we open in order to avoid the introduction of possibilities to execute arbitrary code, eg by allowing unrestricted pointers or function calls.

For precisely the same reason, we also do not want to use features of the CLI that we consider unnecessary and unsafe. In particular, these are all uses of instructions that ECMA 335 describes as not verifiable¹.

In the following, we are going to define precisely the semantics and restrictions of the proposed extensions. All rules and definitions given here are to be understood as an addition to those in ECMA 335. Therefore, the reader must consult the standard, especially regarding specifications of instructions.

4.2 Hardware Access

In principal, every piece of hardware is accessed by reading or writing its interface registers. Depending on the platform, there are two ways of accessing these registers: memory mapping and ports.

In the first way, every hardware register is assigned an address of the address space that contains all memory locations.² Each location can then be accessed using the same CPU operations for accessing memory.

The second way differs by having a special address space for the ports and special but analogue CPU operations for reading or writing these ports. Note, that accessing registers in general can be not quite straightforward since often you have to write a control register before reading or writing a data register which in this case is a mapping of more than one register itself.

4.2.1 Alias Attributes

Because of those similarities to memory access, we introduce two³ new attribute classes: `MemoryAliasAttribute` and `PortAliasAttribute` that can be applied to static and non-static fields. Then, hardware access is represented by the `stfld` and `ldfld` operations of the CLI VES and the respective field. The only difference between both attributes is the generation of different CPU operation codes or operation code sequences by the CIL-to-native compiler. Both attributes have two constructors each, with the address of the hardware location as a parameter of type `int32` or `int64`, respectively.⁴

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple=false, Inherited=false)]
public sealed class PortAliasAttribute : AliasAttribute
{
```

¹see section 3.1.1.3

²Actually, from the CPU's point of view, the memory is not more than another piece of hardware but the special preserving property of memory counts for a lot of analysability of programs. That is why we stick to the common approach of not treating memory as hardware.

³In case there are platforms with more than two semantically diverse address spaces, we need to introduce further appropriate attribute classes.

⁴It would be better to have only one constructor the parameter of which were of type `native int`. But unfortunately, CLS Rule 34 [ECMA 335, p 50] does not allow for such a constructor.

```

public PortAliasAttribute(int port) {}
public PortAliasAttribute(long port) {}
}

[AttributeUsage(AttributeTargets.Field, AllowMultiple=false, Inherited=false)]
public sealed class MemoryAliasAttribute : AliasAttribute
{
    public MemoryAliasAttribute(int address) {}
    public MemoryAliasAttribute(long address) {}
}

```

As indicated by `AliasAttribute`⁵, we call these two attributes *alias attributes*.

4.2.1.1 Exemplified Application to Static Fields

In case of static fields that have built-in types⁶, their use is quite easy to see, since they are conceptually not very much different from global variables in other languages such as C. If a field is attributed with an alias attribute, accesses to it are redirected to the given address in the respective address space. In case of `PortAlias`, the special CPU instructions will be used.

For example, in the following C# declaration of a value type, we find two alias fields for the *data direction register* and the *data register* of port⁷ 4 of the RENESAS H8/3297 series, respectively⁸ [Man H8/3297, p 100]:

```

public struct H8_3297
{
    /* ... */
    [MemoryAlias(0xFFB5)]
    public static byte Port4DataDirectionRegister;

    [MemoryAlias(0xFFB7)]
    public static byte Port4DataRegister;
    /* ... */
}

```

In our C# code, we can access these fields like ordinary fields:

```

/* ... */
H8_3297.Port4DataDirectionRegister |= 0x23;

byte b = H8_3297.Port4DataRegister;
H8_3297.Port4DataRegister = ~0x42;
/* ... */

```

⁵which is just a class for structuring purposes

⁶as defined in [ECMA 335, § 8.2.2]

⁷Note, that in this context the term *port* denotes something different from its usage in *port-based IO*.

⁸In this example, it seems more desirable to have a `MemoryAliasAttribute` constructor that has an `int16` parameter instead of the `int32` one. But as there are no `int16` literals in C# and as all `int16`s are indeed represented by `int32`s there is not much use to such a constructor. Even the objection that an `int32` would reduce the possibility of an erroneous specification does not hold as we could still specify wrong addresses.

And the C# compiler in charge will translate the first line to something very similar to that CLI assembler code:

```
/* ... */
ldsfd [H8_3297]H8_3297.Port4DataDirectionRegister
ldc.i4 0x00000023
or
stsfld [H8_3297]H8_3297.Port4DataDirectionRegister
/* ... */
```

Ideally, an optimising CIL-to-native compiler for the H8/300 processor should be able to translate that code into⁹

```
; ...
mov.b @H'B5, r01
or 0x23, r01
mov.b r01, @H'B5
; ...
```

4.2.1.2 Exemplified Application to Instance Fields

However, the example above is not very well structured, especially when port 5 and 6 of the H8/3297 series are known to have the very same registers as port 4, with the addresses laid out analogously. Thus, we had better write a value type `Port` and change `H8_3297`:

```
public struct Port
{
    [MemoryAlias(0x00)]
    public byte DataDirectionRegister;

    [MemoryAlias(0x02)]
    public byte DataRegister;
}
public struct H8_3297
{
    /* ... */
    [MemoryAlias(0xFFB5)]
    public static Port Port4;

    [MemoryAlias(0xFFB8)]
    public static Port Port5;
    /* ... */
}
```

Here, `DataDirectionRegister` of `Port4` lies at `0xFFB5` and `DataRegister` of `Port5` at `0xFFBA`; the addresses of `Port4` and `Port5` are the base addresses for the members of `Port`. Again, we can access these fields like ordinary fields:

```
/* ... */
H8_3297.Port4.DataDirectionRegister |= 0x23;
```

⁹Besides the usual 16-bit-addressing mode, the H8/300 CPU has an 8-bit-addressing mode for addresses between `0xFF00` and `0xFFFF`. Thus, `@H'B5` is equivalent to `@H'FFB5`.


```
byte b = H8_3297.Port5.DataRegister;
H8_3297.Port5.DataRegister = ~0x42;
/* ... */
```

A C# compiler will turn the first line into CLI assembler code:

```
/* ... */
ldsflda [H8_3297]H8_3297.Port4
ldfld [H8_3297]Port.DataDirectionRegister
ldc.i4 0x00000023
or
ldsflda [H8_3297]H8_3297.Port4
stfld [H8_3297]Port.DataDirectionRegister
/* ... */
```

And as above, an ideally optimising CIL-to-native compiler for the H8/300 processor should be able to translate that code into

```
; ...
mov.b @H'B5, r01
or 0x23, r01
mov.b r01, @H'B5
; ...
```

Hence, the more structured version of that example should not result in any performance loss. We will later see, that the GCC's optimisation passes are indeed that powerful.

4.2.1.3 Formal Definitions

After introducing some definitions for ease of language, we are going to present formal rules that completely specify the use of alias attributes. The first set of rules are restrictions that are necessary in order to keep the concept clear and concise. The attributes' semantics are defined afterwards.

We call a value type a *closed value type* if its instances do never contain any references, directly or indirectly. We can give a crisp recursive definition for that:

Definition 1 *A closed value type is a value type all instance fields of which have closed value types. Built-in value types are closed value types.*

We even do not need the second sentence; all built-in value types do not comprise any instance fields, hence all their instance fields have closed value type. However, to make this point perfectly clear, we did include that sentence in the definition.

In order to understand our second definition, it is a good idea to think of a closed value type as a tree of types: The root node is the value type itself and the first-level children are precisely its instance fields with their types. Likewise, every other node has as its children all its instance fields; if the type of an instance field is a built-in type, then the corresponding node is a leaf. Hence, the leaves represent the actual data whereas all other nodes are merely abstract aggregations of the *same* data.

We will call all nodes of that tree, except for the root node, *components*, ie the instance fields of a type and their instance fields etc. Due to their special function we call the leaves *basic components*. Again, we give a recursive definition:

Definition 2 A component of a closed value type T is either an instance field of T or a component of the type of an instance field of T . A component that has a built-in value type is basic.

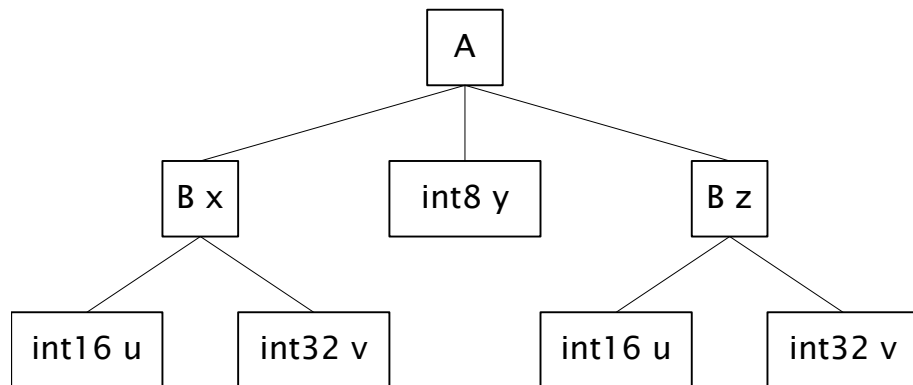


Figure 4.1: Components as a tree

For example, one can find a representation of value type A in figure 4.1:

```

.class valuetype B
{
  int16 u;
  int32 v;
}
.class valuetype A
{
  B x;
  int8 y;
  B z;
}
  
```

Every node below the root node A is a component of A , ie A has seven components but only five *basic* components, the leaves.

Being a bit sloppy, we will also talk about components of a value, which we understand to be the components of the respective type in their actual instantiation as sub-values of that value. In our example, we can see that B has two components but we also say that $A.x$ and $A.z$ have two components each.

4.2.1.4 Formal Restrictions

Obviously, we need uniqueness and validity rules:

Rule 1 *A field must not have more than one alias attribute.*

Rule 2 *The usage of an alias attribute must specify an address that exists in the respective address space on the underlying platform and that is not in use by the runtime for internal purposes such as providing for stack and heap.*

Furthermore, we do not want to allow the application of an alias attribute to a field the type of which is a class type because such a field must be assigned `null` or a pointer to an object that is subject to automatic memory management. But since these fields are not necessarily memory cells and may thus not always *contain* the pointer after the *assignment* of that pointer, there would be no clear semantics for garbage collection. Therefore, we need the restriction to closed value types.

Rule 3 *Alias attributes may only be used on instance fields of closed value types and static fields whose type is a closed value type.*

With the next rule, it is always clear, whether a given structure is an alias or an entity that is to be allocated on the heap. This is important when pushing the address of that structure on the virtual execution stack. It would be necessary to push both addresses, the address of that location for its non-alias-attributed parts and the base address for the alias attributes. This is an unnecessary overhead induced by syntactic sugar as you can always get along by having separate types for aliases and common value types.

Rule 4 *The components of a closed value type `T` must either all be attributed with a `MemoryAlias` or none of them. Accordingly, any static field that has type `T` must be attributed or must be not attributed. An analogous proposition holds for `PortAlias`.*

The next rule targets a similar problem: We forbid the declaration of a static field without an alias attribute if the type of that field contains alias attributes. Suppose, we had this value type `T`:

```
public struct T
{
    [MemoryAlias(0x00)]
    public byte x;

    [MemoryAlias(0x24)]
    public byte y;
}
```

and two static fields `a` and `b` of that type, `a` attributed properly, `b` not. Let `T` have a member function `m`. The implicit instance argument of `m` would be an address of a location of type `T`. When we pass `a` we certainly want to pass the alias address of `a` (in fact we do not have another one), in case of `b` we had to pass the usual address of `b`.

But the layout of T is not the same for a and b ¹⁰, which alone is reason enough for our rule. Moreover, we would need *two* versions of m , one for a and b each (or one method containing m 's code twice). With respect to the limited resources of embedded systems we consider this unacceptable.

Rule 5 *For any closed value type T and any static field f that has type T , the alias attributes must correspond: Iff T contains fields with alias attributes then f has an alias attribute of the same type.*

4.2.1.5 Formal Semantics

For the specification of the semantics, we first detach alias attributed types from the usual memory allocation.

Rule 6 *Types that have alias attributed instance fields are never subject to the runtime's memory management.*

Next, we give a recursive rule for how addresses are obtained. Note, that because of our restrictions above, we will have a static alias attributed field for every non-static attributed field. Therefore, this recursion will always have an end, thus defining the address of every field. Further note, that the recursion is somewhat implicit because it results from the semantics and usage of `ldflda`; whenever there are nesting depths greater than one, say `x.y.z`, the addresses are loaded sequentially by first using `ldsflda` followed by an appropriate number of `ldflda`, in our example (simplified): `ldsflda x; ldflda y; ldfld z`.

Rule 7 *Let x be the value attached to a field f by an alias attribute. If f is static, its address is x . Otherwise, its address is the sum of x and the address of the field that f is a field of. In terms of instructional semantics this means:*

1. `ldsflda f` loads x on the stack
2. `ldflda f` takes an address a from the stack and pushes $a + x$

Finally, we can now give the essential rules for the direct implications of alias attributes. After all, they are straightforward without need for explication. Just note, that due to our restrictions, we can only use alias attributes in conjunction with instance or static fields. This means, we need not specify any accesses to local variables or arguments.

¹⁰If the layout was the same, we would waste quite a lot of memory. In the case of `a.x` and `a.y` having addresses at both ends of the memory space, this waste would be massive for `b`.

Rule 8 Let f be a `MemoryAlias(x)` attributed field the type of which is a built-in type of size s . Then, the memory block of size s starting at the alias address of f is never to be used for memory allocation. Accesses to f are to be redirected to its alias address:

- `ldsfld f` reads from address x and pushes that value on the stack
- `stsfld f` pops a value from the stack and writes it to address x
- `ldfld f` takes an address a from the stack, reads from address $a + x$ and pushes that value on the stack
- `stfld f` takes an address a from the stack, then pops a value from the stack and writes it to address $a + x$

Rule 9 Let f be a `PortAlias(x)` attributed field the type of which is a built-in type of size s . Accesses to f are to be translated to accesses to x using the special instructions of the underlying platform for port-based I/O:

- `ldsfld f` reads from address x and pushes that value on the stack
- `stsfld f` pops a value from the stack and writes it to address x
- `ldfld f` takes an address a from the stack, reads from address $a + x$ and pushes that value on the stack
- `stfld f` takes an address a from the stack, then pops a value from the stack and writes it to address $a + x$

4.2.1.6 Remarks

Our `MemoryAlias` and `PortAlias` attributes are means to tell the compiler about locations, not about values. Therefore, whenever programmers apply these attributes to instance fields of value types they tell the compiler about the structure of locations of that type. While values of a compound value type might still be considered contiguous we give the programmer the opportunity to break up locations of that type and spread their components throughout the address space. This allows for the logical unity of physically scattered entities.

In order to see why we consider these extensions appropriate, recall the notions of *value* and *location* in the CLI: While value types define the representation of values [ECMA 335, § 8.2.3 part I], locations (such as parameter or local variables, or static fields) store values [ECMA 335, § 8.3 part I]. Hence, a location must provide for enough space for a value. Further, it is important to note that values are a *logical* concept whereas locations are a *physical* one. A value may have a different layout than a location of the same type as long as the runtime (or the CIL-to-native compiler) assures the right

mappings. This is because, conceptually, there is no restrictions on *how* values are loaded from or stored into locations [ECMA 335, §§ 3.38, 3.43, 3.61, 3.63, 4.10, 4.14, 4.28, 4.30 part III].

It further complies with the concept of value types [ECMA 335, p 18]:

The values described by a value type are self-contained (each can be understood without reference to other values).

This means, we can think of values of a value type as raw data, thus allowing for interaction with hardware that is not aware of a CLI runtime.

4.2.2 Alternatives

We deliberately did not follow the approach of allowing access to addresses that are determined at runtime such as represented by the methods `ReadByte` and `WriteByte` in the class `System.Runtime.InteropServices.Marshal` of MICROSOFT's implementation of the CLI in the .NET FRAMEWORK, the classes `IoPort` and `IoMemory` of the SINGULARITY project (see section 2.2), or the REAL-TIME JAVA class `javax.realtime.RawMemoryAccess` [RTJS, 2000]. This is merely pointers in disguise that could result in wild pointers bypassing security mechanisms and compromising analysability of programs. Moreover, we do not see any necessity of accessing particular addresses other than for interaction with the underlying platform.

For the same reasons, we revoke the proposal of using `unsafe` sections in C# code [LUTZ, 2003][ECMA 334, p 417]. In particular, the C# standard explicitly states that the possibility of introducing *unsafe* code is to be considered a *safe* feature because the unsafe code is clearly marked so then. This is a rather weak explanation, and therefore we prefer extensions that allow to write safe (in the sense above) code.

Furthermore, our approach has several advantages in comparison to using a proxy. First of all, a proxy had to be generated. This is semantically imprecise as the hardware is not to be created once our programs runs on it; and besides the memory consumption of a proxy object whose only task is to forward access to other locations, the address of the object had to be specified at *runtime* allowing for arbitrary memory access. Additionally, the implementation of proxies is a much harder and thus more error-prone task than our approach.

There are linkers that allow the placement of certain variables at certain memory locations. Our memory alias attributes could have been modeled using this mechanism; instead of applying an attribute to the field in question, one declares the field as external and defines its address in a linker script. Our approach has the advantage of providing a uniform mechanism for accessing hardware as well as means for suitable structural abstractions. Additionally, linker scripts are linker specific, whereas with our attributes the OEMs have to specify their hardware only once.

In ADA, it is possible to declare addresses of variables in the source code [ISO/IEC 8652:1995]. This is essentially the same as our `MemoryAlias` applied to a static variable. There is no equivalent for port-based I/O. As with linker scripts, our approach is more uniform and allows for better modularisation.

We are aware that our approach does not allow for *every* speciality that could arise from interacting with hardware. If, for example, a memory-mapped register can change its address arbitrarily at run-time there is no way to deal with this using our fixed-address attributes.¹¹ However, the advantages (analysability, safety and readability/maintainability) overcome these restrictions easily, especially when considering that such specialities are rather rare in embedded systems.

Moreover, we did *not* include an implicit definition: If a type contains fields attributed with `MemoryAlias` then a field of this type, that does not bear a `MemoryAlias`, could have an implicit `MemoryAlias(0x00)` attribute, thus guaranteeing a base address for calculation of absolute addresses. For consistency reasons, we decided against this.

There were suggestions to restrict the use of both alias attributes to static fields only and to use two attributes `MemoryAliasOffset` and `PortAliasOffset` for instance fields because it was claimed that it might be an advantage for persons who do not want to spend too much time on learning the whole concept. It was argued that it might not be intuitively clear that the original attributes applied to static fields have a different effect as if applied to instance fields. On the other hand, this would sacrifice the minimalistic approach, introducing another construct where it is not absolutely necessary. Since an alternative implementation is not very hard to realise¹², we decided to leave this until it actually turns out to be necessary.

4.3 Interrupt Handling

Originally, we intended to integrate the interrupt handling directly into the CLI's event mechanism. But there is a big difference between events in the CLI and interrupts: An event must provide for two methods for adding and removing event handlers. It is then possible to have multiple handlers for the very same event. On the other hand, at the hardware level, there is usually space for registering only one interrupt handler that is called on each occurrence of that interrupt.

Therefore, in order to prepare the ground for integration into the CLI's event mechanism, our approach models the separation of interrupt handlers in immediate and scheduled interrupt services that is often found in modern operating systems [LIU, 2000, pp 504–507]. However, we will not describe concepts for scheduled interrupt services since

¹¹If in contrast there are only a limited number of such memory locations, we could specify all of them and then use the appropriate one.

¹²Essentially, the only thing you have to do is to check for wrong attribute usage which is quite easy a task to perform.

these can be sufficiently modeled with standard CLI techniques such as events; we shall rather concentrate on the parts dealing directly with hardware.

4.3.1 Immediate Interrupt Handler

When an interrupt occurs, the CPU saves the immediate context (such as program counter and condition registers) on the stack and calls a function the address of which is obtained from an interrupt vector. In contrast to calls of common functions that push just the program counter on the stack, interrupt handling functions need a different return instruction that makes the CPU restore the immediate context. We state more precisely what we mean by *immediate context*:

Definition 3 *The immediate interrupt context (IIC) is that part of the CPU that is saved automatically on the stack when an interrupt occurs. The immediate working context (IWC) are all other registers of that CPU.*

In order to support immediate interrupt handling, programmers need two means of expression. First, they must be able to tell the compiler to save the IIC and the IWC on function entry and to restore them on exit. For handling the IIC, the compiler will then usually emit the return-from-interrupt instruction instead of the usual return instruction for that platform. Concerning the IWC, it is often preferable not to save all registers but only those that are used by the interrupt handler. Since this might increase performance by reducing interrupt latency, we allow the compiler to do so.

For that purpose, we propose the usage of an attribute `InterruptHandler` defined like that:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = false)
]
public sealed class InterruptHandlerAttribute : Attribute
{
    public InterruptHandlerAttribute() {}
}
```

Rule 10 *If the attribute `InterruptHandler` is applied to a method `m`, the compiler must make sure that the immediate interrupt context is restored on exit of `m`. Furthermore, any parts of the immediate working context that are manipulated by `m` must be saved on entry and restored on exit of `m`.*

4.3.2 Static Method Delegates

Second, programmers need to assign the address of a function to an interrupt vector. The only way to deal with function pointers in high-level languages such as C# is passing a function pointer to the constructor of a delegate. Therefore, we will need to use delegates.

Unfortunately, a delegate is a rather sophisticated construct for our purposes. It is of a type which inherits from `System.Delegate`, and thus `System.Object`. Therefore, it is subject to memory management and contains every overhead `System.Object` induces. Moreover, delegates are intended to wrap static as well as instance methods hence making a field necessary that contains a reference to the instance they are linked to. Contrarily, the interrupt vector will usually not provide more space than needed to fit in a pointer.

For that reason, we introduce the attribute `StaticDelegate`:

```
[AttributeUsage(AttributeTargets.Delegate, AllowMultiple = false, Inherited =
    false)]
public class StaticDelegateAttribute : Attribute
{
    public StaticDelegateAttribute() {}
}
```

This attribute changes the semantics of a delegate *type* to value-type semantics. While this seems a strange thing to do, be aware of the general concept of value types in ECMA 335. Each value type defines two types: the value type itself and a class type suitable for boxing instances of that value type. When applying `StaticDelegate` to a delegate we do precisely the same; we define a value-type delegate and also a class type. The only problem here is that such a class type for a `StaticDelegate` value type does not inherit from `System.ValueType` but `System.Delegate`. It is, however, uncertain if this has any major impact on the CLI but we win a lot. Fields of that type can now be defined to have precisely the size of a pointer on the underlying system. Along with alias attributes, we are now able to specify interrupt vectors in C#.

Rule 11 *A delegate type D with signature S that is attributed `StaticDelegate` is a value type comprising a single field of type `native int`. Only a limited number of operations are specified for such a type:*

1. *The constructor pops two arguments from the evaluation stack, the first of type `System.Object`, the second a token referring to a compatible static method as defined in ECMA 335 Partition II § 14.6.1. When called, the first argument is always ignored and should be `null`; the second argument is put on the stack (converted to D if the compiler keeps track of such types) on return.*
2. *An instance method `Invoke` with signature S. On call `D.Invoke(S)` or `callvirt D.Invoke(S)` an operation equivalent to `calli S` is performed.*
3. *`ld*` and `st*` as defined in Partition III.*

Now, in order to use a method as an interrupt handler, we first have to create a delegate wrapper object for this method¹³ and then assign this to the alias field for the interrupt vector. With that last rule, there is no problem; we can easily avoid the indirection of a delegate and use the method's address directly.

¹³In C# 2.0, this can be done implicitly when assigning.

4.3.3 Exemplified Application

Combined, these two constructs allow for the use of interrupts. We will demonstrate this with an example. For the Renesas H8/3297 series, the interrupt vector for the non-maskable interrupt (NMI) and the serial timer control register (STCR) could be specified like this:

```
[StaticDelegate]
public delegate void InterruptHandler ();

public struct VectorTable
{
    /* ... */
    [MemoryAlias (0x06)]
    public InterruptHandler NonMaskableInterrupt;
    /* ... */
}

public struct H8_3297
{
    [MemoryAlias (0x0000)]
    public static H8_3297 Board;

    [MemoryAlias (0x0000)]
    public VectorTable VectorTable;

    /* ... */
    [MemoryAlias (0xFFC3)]
    public byte SerialTimerControlRegister;
    /* ... */
}
```

We can now give a simple¹⁴ example how the STCR will be manipulated on occurrence of the NMI.

```
using Hardware;
using Renesas;

namespace TestRcx
{
    public class MainClass
    {
        [InterruptHandler]
        static void Handler ()
        {
            H8_3297.Board.SerialTimerControlRegister = 123;
        }

        public static void Main ()
        {
            H8_3297.Board.VectorTable.NonMaskableInterrupt = Handler;
        }
    }
}
```

¹⁴but mind- and useless

should be translated to

```

__ZN14test_interrupt7TestRcx9MainClass7HandlerEv:
    mov.w r2,@-r7
    mov.b #123,r2l
    mov.b r2l,@-61:8
    mov.w @r7+,r2
    rte
.global _main
_main:
    mov.w #__ZN14test_interrupt7TestRcx9MainClass7HandlerEv,r2
    mov.w r2,@6:16
    rts

```

Note, that the strategy of setting interrupts directly might not be pursued if the underlying platform is managed by an operating system because it probably discourages such a course of action. However, the operating system will provide for a function such that you can use the common ways of accessing external functions.

4.3.4 Remarks

The introduction of the `InterruptHandler` attribute was inspired by the GCC attributes `interrupt/interrupt_handler`. Internally, this attribute is simply translated to its GCC counterparts. The only alternative we can see to our approach to interrupt handling is the usage of custom, platform-dependent runtime functions.

4.4 Concurrency

4.4.1 Context Switches

There are some important I/O related tasks programmers have to perform when implementing concurrent mechanisms most of which are related to context switches. In order to switch a context in non-cooperative multitasking systems, an interrupt is needed the handler of which saves the current task's immediate working context and establishes the next task's IWC. Thus, when the interrupt handler returns, the CPU automatically continues operation with the next thread.

However, while we can generally describe this process in a few words, it is, naturally, very platform dependent. The most elegant way we can think of is to provide for an attribute `ContextSwitching` that can be attached to methods with a specific signature; they must be static, declare an argument for the context pointer, and return the next context pointer. The compiler shall then generate code that fills the argument slot with the appropriate pointer to the saved IWC before any other code is executed¹⁵ and that sets the new context once the method returns. Figure 4.2 shows the stacks when the sequence is executed.¹⁶ The first picture shows that the argument and return slots must

¹⁵But see remarks regarding attribute `Uninterruptible`.

¹⁶Stacks grow from top to bottom, arrows indicate pointers.

be available on the stack, whereas the second and third ones show the stack before and after execution of the method's real body.

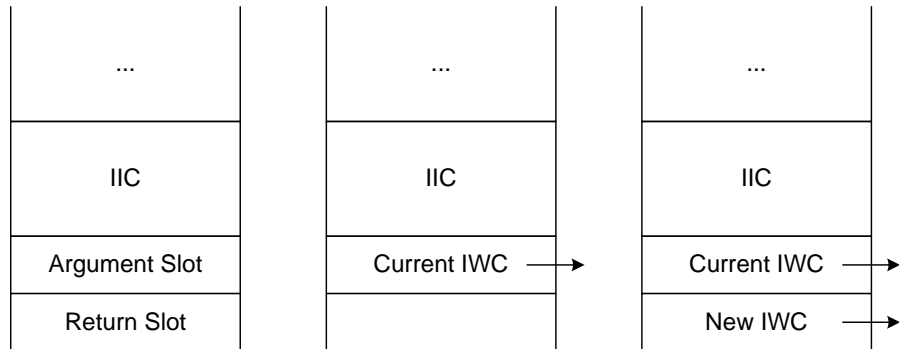


Figure 4.2: Stacks when switching the context

A simple strategy for implementing context switches is to push the IWC on the stack and then to simply switch the stack pointers. This can be realised with our approach, too, as figure 4.3 displays for two contexts 1 and 2. In the first picture, you can see the IWC pushed on top of the argument and return slots. After that, the current stack pointer (Stack Pointer 1) is written in the argument slot and the method's body is executed. When the method's body is done, it should have put the Stack Pointer in the return slot, such that the compiler-generated epilogue can set the stack pointer to Stack Pointer 2. Now, the CPU uses the stack of context 2.

Nevertheless, the whole approach could turn out not to be quite easy to implement on architectures that do not use registers for passing arguments. In that case, the caller usually reserves argument and return slots on the stack before the return address that is pushed on the stack when the call occurs. As you can see in figure 4.2, the slots *must* be placed after the return address just because the return address is part of the IIC (the program counter of the context the interrupt interrupted). Therefore, the compiler must be aware of such a different stack layout, which can require a lot of effort.

For that reason, we present a slightly less elegant version. The context-switching method must have the same signature as above but it need not be attributed. Instead, there is a class `Hardware.Cpu` with a static method `void InvokeScheduler ()`, a delegate type `SchedulerDelegate` attributed `StaticDelegate` and with the appropriate signature for the context-switching method, and a static field `Scheduler`. Now, when `InvokeScheduler` is called it saves the IWC, provides for the argument and return slots at the right position on the stack, and calls the method in field `Scheduler`. When that method returns, `InvokeScheduler` reads out the return value and sets the new context.

In both version, we further need a static method `Hardware.Cpu.InitializeContext` that initialises a new context. Additionally, we propose the static method `Hardware.Cpu.Sleep` that a compiler shall replace by the respective power-saving instruction or sequence of

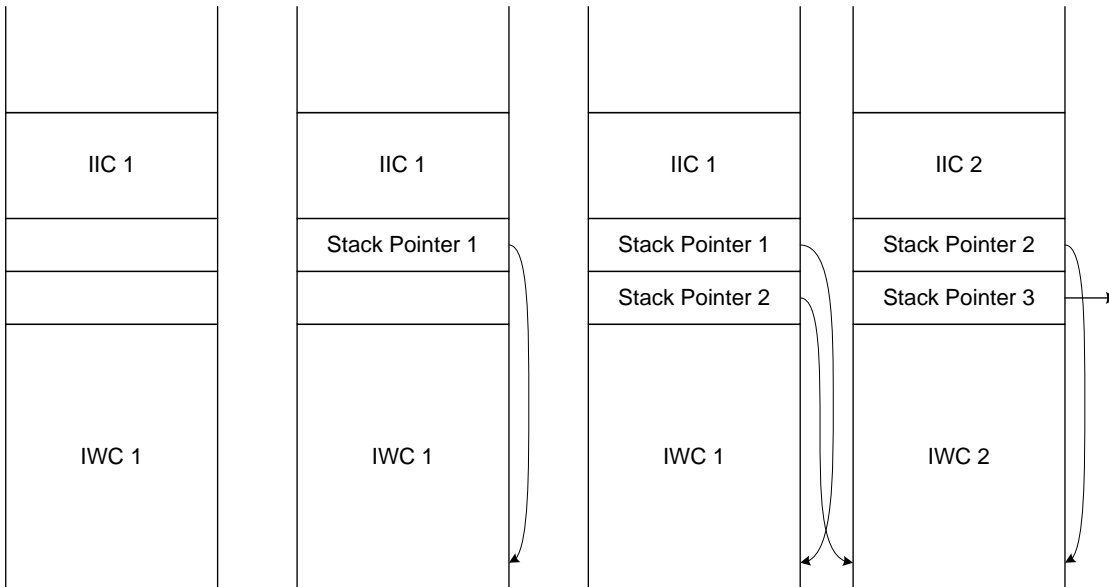


Figure 4.3: Contexts on the stack

instructions of the underlying hardware.

With these extensions, arbitrary schedulers could be hooked in a timer interrupt handler like that:

```
class MainClass
{
    static void Main ()
    {
        Timer.Init ();
        Timer.Scheduler = Scheduler.DetermineNext;
        Timer.Start ();
        while (true) Cpu.Sleep ();
    }
}

public class Timer
{
    public static Cpu.SchedulerDelegate Scheduler;

    public static void Init ()
    {
        /* ... */
        Cpu.Scheduler = Handler;
        TimerInterruptVector = Cpu.InvokeScheduler;
        /* ... */
    }

    public static Cpu.Stack Handler (Cpu.Stack stack)
    {
        /* ... */
    }
}
```

```
        return Scheduler (stack);
    }
}
```

In this example, the (fictitious) `TimerInterruptVector` is set to `Cpu.InvokeScheduler`, hence that method is called whenever the timer interrupt occurs. Since it calls `Timer.Handler`, the scheduler (`Scheduler.DetermineNext`) is finally invoked whenever a timer interrupt occurs.

4.4.2 Disabling Interrupts

We need to bring up the issue of interrupt disabling/enabling, which is also a very platform dependent task. This is particularly true for disabling and enabling *single* interrupts. However, because at certain points—for example when implementing synchronisation primitives—we have to disable all interrupts (if possible), we propose two alternatives.

First, two static functions `Hardware.Cpu.Interrupts.DisableAll` and `Hardware.Cpu.Interrupts.EnableAll`, that simply disable or enable all interrupts. These functions are intended to be the minimal wrapper for disabling or enabling interrupts, ie without support for nested interrupts etc. OEMs might want to integrate the enabling and disabling of single interrupts in this class.

Second, an attribute `Uninterruptible`, that can be attached to methods. If so, the very first instruction of a translated function shall disable all interrupts and its very last instruction shall enable them. Such methods are necessary for immediate interrupt handlers that shall not be interruptible by other interrupts, for example when implementing synchronisation primitives. Usually, such instructions do not affect the IWC, so they shall be involved even before the instructions that save the IWC in context switching methods.

4.4.3 Remarks

Probably, people would like to hide the context switch behind a platform dependent, static property `Hardware.Cpu.CurrentContext`. Context switches must then include a call to the getter method to save the current context and a call to the setter method to set the next context. Then we could write code like this:

```
using Hardware;
using Renesas;

class Scheduler
{
    public static Context currentContext;

    public static void TimerHandler ()
    {
        Cpu.CurrentContext = Scheduler.determineNextThread (Cpu.CurrentContext);
    }
}
```

```
}  
}
```

We decided against this because this would imply a freedom that the programmer does not have. The context must always be read in at the beginning of a method and can only be set at its end.

4.5 Summary

We presented new extensions to the standard ECMA 335. First of all, we showed a general approach for mapping hardware registers not only to simple variables but to structured value types for both, memory-mapped and port-based I/O. The introduction of appropriate attributes and an enhanced redefinition of value types semantics and related instructions, enables OEMs to specify their hardware in high-level languages such as C#.

Further, we seized and formalised the idea of attaching an attribute to a method in order to make the compiler generate code suitable for immediate interrupt handlers. By reducing the generality of delegates, our concept of “static delegates” allows for the integration of external locations for function pointers with the CLI. With both mechanisms, it is possible to write the whole interrupt handling in high-level languages.

Lastly, we were putting things forward towards the implementation of an operating system by introducing a general concept for context switches. Separating scheduler, timer interrupt handler, and context switch, this concept is a flexible means for implementing many different kinds of concurrency.

Chapter 5

Implementation

Having introduced exciting new extensions in the last chapter, we are now going to show an implementation. To begin with, we present the front end that we developed for the GCC. A description of the implementation of the extensions themselves follows immediately before the display of our performance evaluation. Note, that the generated code is evaluated in greater detail in the next chapter.

5.1 Front End Overview

5.1.1 Structure

Due to the nature of its task, our front end is quite different from other front ends for the GCC. Instead of parsing text, we read binaries, that another compiler (eg MICROSOFT's `csc` or MONO's `mcs`) has emitted. Therefore, many complex tasks that parsing involves are utterly unnecessary in our case. The three most important ones are:

1. Identification of entities by parsing their name—every entity is represented by a binary token, the name of an entity is to be seen as an attribute rather than a referential element.
2. Determination of relations between entities—most of these are given in a straightforward manner by using reference tables associated with entities.
3. Processing of program code structures—each method is a flat sequence of instructions; locals, and parameters are known in advance.

Moreover, in order to further simplify our task, we use code from the DOTGNU PORTABLE.NET project that we took from their freely accessible CVS versioning system repository¹, namely the `pnet/include` and `pnet/image` subdirectories as well as parts of the `pnet/support` subdirectory. There is a `pnet` subdirectory in our front end's directory containing these files.

This code reads in one or more binary CLI files and builds up fully linked compile-time structures for all entities in those files using a kind of light-weight object orientation typical for C: If a type `struct A` is extended to a type `struct B` the first element of `struct B` is of `struct A`. Casts from one type to the other are then safe and do not need any runtime handling. For all program items there is a struct type `ILProgramItem` which is the root for the type hierarchy.

On the other side, the GCC's optimisation and code emitting routines expect the front end to represent whatever program it reads as a graph² that is made up of nodes

¹<http://cvs.savannah.gnu.org/viewcvs/?root=dotgnu-pnet>

²Often called a tree, which might be misleading because there are some references which spoil that. Using the term *graph*, we are on the safe side.

of type `tree`³. Thus, our front end's task is the translation of one graph made up of `ILProgramItem` nodes into another one made up of `tree` nodes. In essence, the general structures of both graphs are equivalent for the parts that we are interested in, thus making the translation quite straightforward. The hard part is to find out the sparsely documented internal concepts of the GCC.

During the translation process, we need to keep track of which `ILProgramItem` was mapped to which `tree`. In contrast to the `LEGO.NET` front end, we do not use the GCC's internal hashtable of identifiers for that purpose⁴. Instead, we added an element to `ILProgramItem` allowing us to store a reference to the respective `tree`. Nevertheless, we do store our identifiers in the internal hashtable to avoid having the GCC internal garbage collection remove the corresponding `tree` objects.

Besides several hooks that are intended to be called by GCC routines for command line options handling, type conversions and similar tasks, the front end essentially consists of three layers (see figure 5.1):

1. a main layer that builds most of the `trees`
2. the runtime specific layer (RSL) that builds `trees` that are runtime specific
3. the CIL reader adaption layer (CRAL) that connects the main layer to the `DOT-GNU` code

This general structure allows for easier replacement of the runtime and the external code that reads in the binaries. It further clarifies concepts by separating them.

Following the GCC's execution model for front ends, the main layer is first pre-initialised, before it handles front-end specific command line options. After that, the main layer's initialisation routine is called, which initialises the other layers as well. It is here that the runtime specific system library `mscorlib` is loaded into memory. Once initialised, the main layer calls the CRAL's `cral_parse_file` function, which loads the file that is actually to be parsed, finds this file's entry point and starts parsing the respective method.

5.1.2 Parsing and Translation of Methods

Methods are parsed one at a time; when the compiler finds out that a method `m` has to be compiled—eg because it is the target of a method call or because it is a virtual method—it stores a reference to `m` in the deferred-methods queue. As soon as the parsing of a method has been completed, the next method of that queue will be parsed unless it has already been compiled. This process is repeated until there is no method left in

³Actually, `tree` is a pointer type for `tree_node`. However, since the latter is only used at very special places we stick, for readability's sake, to the (common) term `tree`.

⁴For a few exceptions see how we realised labels and built-in methods.

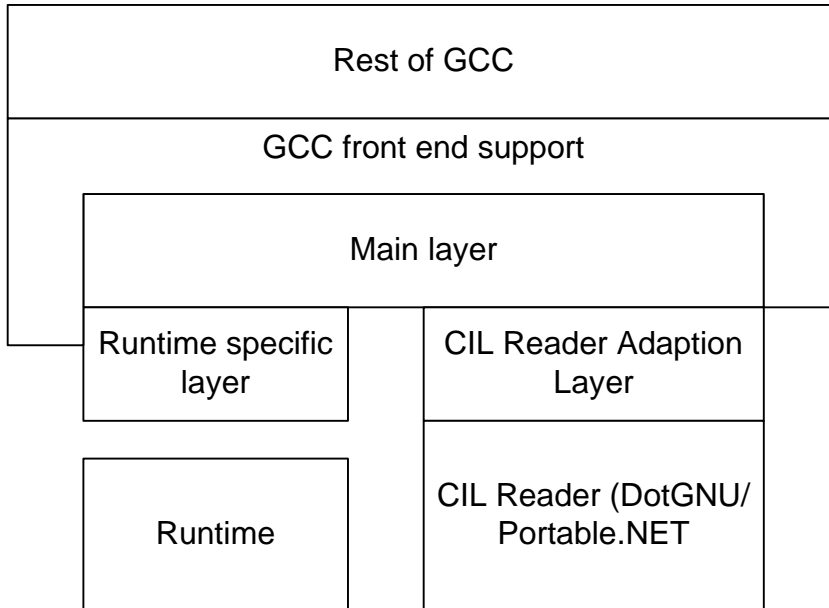


Figure 5.1: Structure of the front end

the queue. We can then be sure that all methods that can be reached by the entrypoint have been compiled.⁵

This mechanism allows for easy integration of `mcorlib`'s system code with the application once we support all constructs that are used by the system library. Until then, we need a special runtime that contains implementations of those methods and types that we cannot compile but that applications need. Additionally, the RSL must announce those functions to the front end as being external. Actually, all methods that the front end encounters are *registered* but only those are *parsed* that are not marked as being external either by the CIL signature of the method itself or by the RSL. The main advantage over `LEGO.NET` is that we do not need to build our own lightweight `mcorlib`; we can use any `mcorlib` that we have an adjusted runtime for.

When registering a method, the `tree` representation of its type is built along with the method's full name and mangled name (see section 5.1.4). Out of this, a `tree` for the method itself is built, which is then stored in the method's `DOTGNU` representation for further use.

When a method is going to be parsed, that `tree` is supplemented by the parameters and local variables the method declares. Additionally, the evaluation stack is prepared that will be used for the virtual execution of the method. Then, the opcodes that form the body of the method are parsed sequentially; we never need to follow branches.

⁵Since reflection is currently not implemented there is no need for special provisions regarding reflection.

Before each opcode is parsed we set a label in the instruction list of the `tree` representation that corresponds with the current program counter in the opcode stream. With that, we can easily translate unconditional branches to `GOTO_EXPR` and conditional branches to combinations of `COND_EXPR` and `GOTO_EXPR` `trees` the target of which is that label. Unnecessary labels are later removed to avoid too much unimportant information in debug and assembler output.

We have an array that is 256 bytes large and contains the addresses of handlers for the simple opcodes, ie the ones that are one byte in size. Thus, instead of using a huge switch construct, we have a clear and concise way of organising our code that further results in efficient compile-time behaviour. The handler for the extended opcodes—two-byte opcodes starting with `0xFE`—uses the same mechanism for the second byte.

5.1.2.1 Evaluation Stack

Our evaluation stack consists only of `trees` representing variables local to the current method, in contrast to the `LEGO.NET` front end that allows all `trees` which are expressions. These variables are created in addition to the locals the method already has. The pair *type* and *stack height* determines the stack variable to be used, ie whenever a value of the same type is pushed on a stack with the same height the same variable will be used.

`trees` for variables are build when needed and stored in additional arrays for reuse. We need such an array for every type that we track, and their size is determined by the maximum stack height each method has to declare [ECMA 335, § 1.7.4 part III]. Further, the standard allows instances of arbitrary value-types to be pushed on the evaluation stack. Since we cannot know which types these might be *before* parsing the method, we included an array for value types that contains a *chain* of variables for every stack height. Whenever a value is pushed on the stack that does not fit into the other arrays it is added to the respective chain. Conversely, when a stack location is accessed, the right variable is searched for in the chain. This search is a simple linear one as it should not happen to often that you have lots of different value types at the same stack location.

Moreover, we track the types of integer stack locations in more detail than required by ECMA 335⁶ in order to allow for better optimisation by the GCC. This includes conversion of constants to smaller types if they fit in. The GCC can expand constants to bigger type if necessary; the other way round causes more trouble.

Whenever a forward branch occurs, we need to copy the whole stack because it may happen that the stack is not empty and the target follows an unconditional branch. For example, consider this code fragment in C#:

```
struct A
{
    int x;
```

⁶§ 12.3.2.1 part I clearly states we may do so.

```
public int M (bool b)
{
    x = b ? 12 : 34;
}
}
```

MICROSOFT's `csc` translated the body of `M` to

```
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  brtrue.s IL_0008

IL_0004:  ldc.i4.s 0x22
IL_0006:  br.s IL_000a

IL_0008:  ldc.i4.s 0x0c
IL_000a:  stsfld int32 A::x
IL_000f:  ret
```

After `IL_0002` the evaluation stack consists of the value of argument 0 (`b`), which is of type `bool` and so it does immediately before `IL_0008`. On the other hand, the stack consists of the value `0x22` of type `int32` after execution of `IL_0006`. With our sequential parsing, we would use that stack when virtually executing `IL_0008` if we did not copy the right stack from `IL_0002`. We would then actually access the wrong stack variable, ie the one of type `bool` instead of `int32`.

This problem cannot occur for backward branches, say to a location x . Either we already encountered a forward branch to x , or the standard requires that the stack be empty at x [ECMA 335, § 1.7.5 part III].

5.1.3 Translation of Classes

Similarly, classes are parsed when necessary. Instance fields are always parsed as a whole whenever an object of that class is used. This is because at certain points the GCC needs the final size of a type, such that adding fields incrementally is not possible. Thus, in order to get rid of unused fields we would need to compile everything twice. Static fields are parsed separately from each other because internally they are represented as global, independent variables. Hence, if a static field is never referenced it will not be used in the resulting binary.

For runtime creation of objects (as requested by the CIL's `newobj` instruction), the main layer calls a method in the RSL. The current RSL generates a call to a method `newobj` that is expected to be linked to the resulting object file.

5.1.4 Name Mangling

Using the GCC, programs written in high-level languages can be translated to assembly language for supported platforms. Since these assembly languages usually have very limited support for identifiers, more sophisticated naming concepts have to be mapped.

This includes nested, overloaded, and overridden identifiers. Fortunately, there exist solutions for this problem, that we just had to implement for our front end. In accordance with GCC's C++ front end [GCCc], we implemented an appropriate subset of the specification given by CODESOURCERY LLC [CSABI, 2001].

5.2 Extensions

Due to our considerate definitions *and* the structure of our front end, all extensions could be implemented without great effort.

5.2.1 Alias Attributes

If a (static or instance) field is attributed `MemoryAliasAttribute`, the front end adds a `tree` node to it (the *alias tree*), representing an indirect access to the given address, ie an `INDIRECT_REF` node with an integer-constant node (the address) as argument. Whenever the front end pushes an expression on the stack, it checks whether there is such an alias tree associated with the expression. Mostly, that alias tree is used instead of the expression itself—when pushing something on the stack or popping into something from the stack which is not an instance field. In case of of instance fields, an alias tree is popped from the stack and a new alias tree is produced, which is a `PLUS_EXPR` the operators of which are the alias tree from the stack and the alias tree associated with the field.

One very important thing is the right use of the `TREE_VOLATILE` and `TREE_STATIC` flags, that can be set on any expression. If both are set, the middle end will not optimise accesses to them, which is necessary given the differing behaviour of memory mapped devices. On the other hand, we clearly do not want to mark any expression as volatile, in order not to inhibit useful optimisations. As a short example, look at this (not very sensible) piece of C# code for the Renesas H8/300:

```
using Renesas;
namespace Test
{
    public class MainClass
    {
        public static byte X;

        public static void Main ()
        {
            H8_3297.Board.Timer16Bit.ControlStatusRegister = 0x01;
            H8_3297.Board.Timer16Bit.ControlStatusRegister = 0x11;

            X = 0x01;
            X = 0x11;
        }
    }
}
```

Our compiler translates this to:

```

__ZN10test_alias4Test9MainClass4MainEv:
  mov.w r6,@-r7
  mov.w r7,r6
  mov.w #-111,r2
  mov.b #1,r31
  mov.b r31,@r2
  mov.b #17,r31
  mov.b r31,@r2
  mov.b r31,@__ZN10test_alias4Test9MainClass1XE:16
  mov.w @r7+,r6
  rts

```

While the first assignment to `X` is omitted, it is not for the memory alias `ControlStatusRegister`.

5.2.2 Interrupt Handling and Multitasking

The attribute `InterruptHandler` could be directly mapped to the GCC attributes `"interrupt"` and `"interrupt_handler"`. Unfortunately, the GCC's back ends are not consistent in the use of these attributes; some use `"interrupt"`, others `"interrupt_handler"`, and there are also some back ends that do neither. For that reason, we simply attach both attributes to such functions, hoping that respective back ends will be supplemented by either attribute in the future.

If a type is attributed `StaticDelegate` we use simple pointer types. The construct was defined in a way that allows to omit type checks at runtime, so we can use function pointers in lieu of more complicated structures.

For the implementation of runtime methods that are part of our extensions consult the examples in the next chapter. Note, that all these methods have to be made known to the compiler. We did so by feigning them as built-in methods. Probably, these methods will be built-in methods in the future.

5.3 Status

By now, the front end deals with the following constructs and instructions:

- complete integer arithmetic
- conversion operations
- all branch and comparison instructions
- standard, instance, virtual and indirect method calls
- value types

- simple class types, inheritance
- strings
- memory alias attributes
- “static” delegates
- interrupt handlers

5.4 Performance Evaluation

When presenting new features of a compiler, qualitative performance analysis, ie the evaluation of the code it produces, is of utmost importance in order to valuate the compiler’s suitability for working environments. We do so when presenting some examples in the next chapter.

Contrarily, getting insight out of a quantitative performance analysis is a bit tricky in this context, as the functionality of this compiler is not quite comparable to other compilers. First of all, we cannot exhibit a complete implementation of ECMA 335 but only a prototypical one. But with increasing functionality of a compiler its temporal performance must degrade except for those rare cases where you can avoid huge conditionals, as we did with the function pointer arrays for the instructions. On the other hand, we introduced constructs that no other compiler understands by now.

Therefore, we have to resort to a C compiler for comparing the performance in case of the constructs we introduced. There is nothing as self-evident as using the GCC with its C front end for that purpose. Unfortunately, we are comparing apples and oranges due to the fundamental differences in the way C and C#/CLI programs are compiled⁷. Each C translation unit—usually a source file with all included header files—is compiled into one file containing platform specific assembly code and further into one object file containing the respective machine code. After that, all translation units are linked into the final binary, binding libraries either now or at program start time. In contrast, the C# files are all compiled into one CLI assembly file which is then further translated by our front end and the rest of the GCC into one platform specific assembly file. This file is translated into an object file that is finally linked against the runtime’s object files.

Because of these differences, our analysis must span all the steps mentioned above. Unfavourably, this involves quite a few pieces of software that are beyond our control, namely the C# compiler, the assembler, and the linker.

The analysis was conducted on a machine containing an AMD ATHLON 2000 and 512 MB RAM, using CYGWIN on MICROSOFT WINDOWS XP. For the C# compiler we used the one included in MICROSOFT’S .NET framework version 2.0.50727. Assembler and linker

⁷The ways presented here are the typical ones; we are aware there are others.

are taken from the GNU BINUTILS collection version 2.16.1. We were using the following two shell scripts⁸. The first one for the C# variant:

```

/.../csc.exe /t:library /out:CliHw.dll "...\\CliHw\\*.cs"
/.../csc.exe /r:CliHw.dll /t:library /out:H8_3297.dll "...\\H8_3297\\*.cs"
/.../csc.exe /r:CliHw.dll /t:library /r:H8_3297.dll /out:RCX.dll "...\\RCX\\*.cs"
/.../csc.exe /r:CliHw.dll /r:H8_3297.dll /r:RCX.dll "...\\test_rcx.cs"
gcc/scil -O3 test_rcx.exe -syslib /.../mscorlib.dll
/usr/local/bin/h8300-hms-as test_rcx.s -o test_rcx.o
/usr/local/bin/h8300-hms-ld --script main.lds --relax --strip-all -nostdlib -o
test_rcx.bin runtime.o test_rcx.o rt.o

```

As you can see, we first compile the CLI libraries `CliHw.dll`, `H8_3297.dll`, and `RCX.dll`, that reflect our separation of concerns. While `CliHw.dll` contains the necessary declarations for the concepts presented in chapter 4, `H8_3297.dll` comprises the definitions for all registers of the H8/3297 board as in the respective reference manual [Man H8/3297]. On top of that, `RCX.dll` declares the memory locations of the deferred interrupt vectors.⁹ Further, the files `timer.cs` and `sound.cs`, which deal with timer and sound, are put in that library as well. We then make `csc` compile our test C# file, which subsequently is fed into our front end using maximum GCC optimisation (`-O3`). Finally, the output is assembled and linked against our runtime files.

In contrast, the C variant uses no libraries. Any code similar to `CliHw.dll` and `H8_3297.dll` has been put into header files to model the modularisation of our CLI example. The files `sound.c` and `timer.c` correspond to the analogously named C# files. They, and the actual test file `test_rcx.c`, are compiled with maximum optimisation, too, then assembled and finally linked:

```

gcc/cc1 -O3 -o sound.s .../c/sound.c
gcc/cc1 -O3 -o timer.s .../c/timer.c
gcc/cc1 -O3 -o test_rcx.s .../c/test_rcx.c
/usr/local/bin/h8300-hms-as test_rcx.s -o test_rcx.o
/usr/local/bin/h8300-hms-as timer.s -o timer.o
/usr/local/bin/h8300-hms-as sound.s -o sound.o
/usr/local/bin/h8300-hms-ld --script cmain.lds --relax --strip-all -nostdlib
-o test_rcx.bin runtime.o test_rcx.o rt.o sound.o timer.o

```

We conducted both tests 4000 times, which can be considered a test base large enough for confident results. Times were taken by the `BASH` `time` directive which is a built-in command into the shell. With respect to the dimension of our measurements, this is sufficiently precise. For evaluating the results, `MICROSOFT EXCEL 2003` was used. The experimental average execution times were 4.63 for CLI/C# and 6.48 for C with experimental standard deviations of 0.064 and 0.099, resp. The minimal and maximal values were 4.53 and 4.98 compared to 6.21 and 6.94. A graphical display of the frequency distribution of execution times for all runs can be found in figure 5.2.

⁸For readability reasons, paths are shortened to `...` where appropriate. In order to stick to the differences, files that are common to both had been made before.

⁹The immediate interrupt handlers residing in the ROM do nothing else than calling the methods these vectors point to, thus providing a means for using custom interrupt handlers.

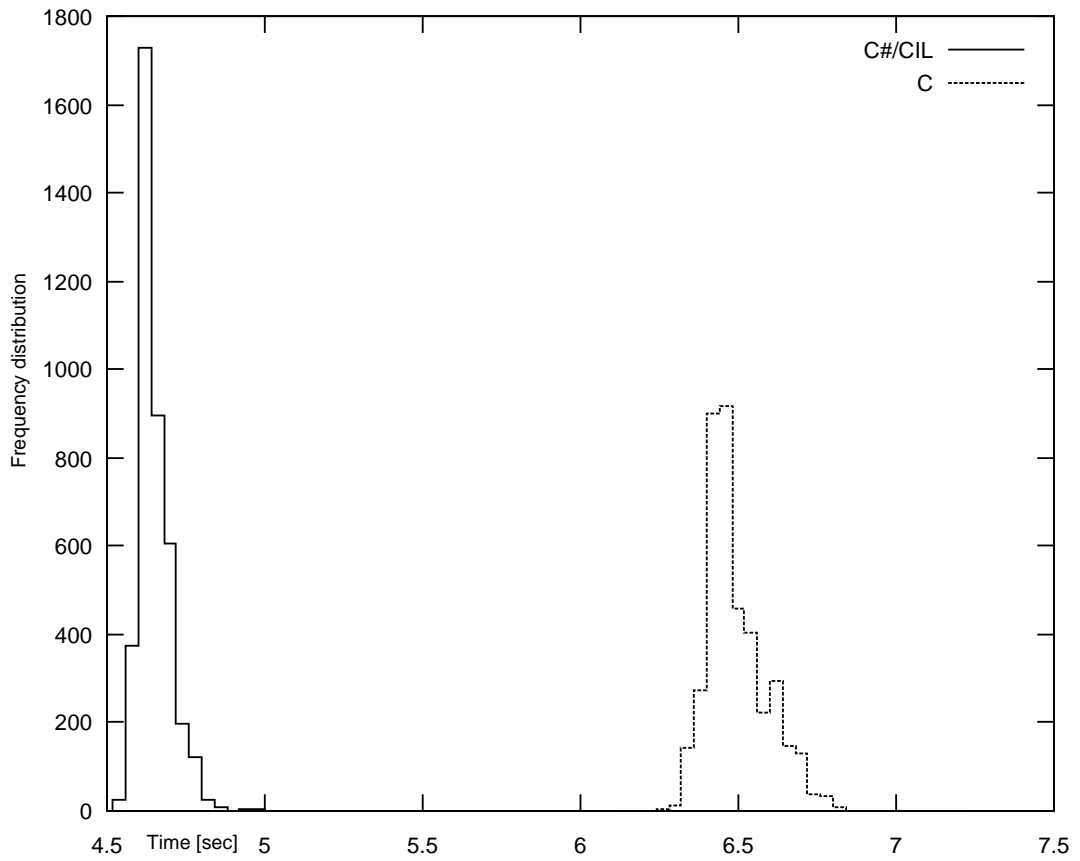


Figure 5.2: Frequency distribution of execution times of test runs

What we can learn from that picture is simply that our compiler is not extraordinarily slow for a practical example. Actually, the performance for the whole process is better than for the C variant, especially when it is known that other examples produce similar results. However, one should not over-estimate the value of such observations. More interesting are the following results regarding the *complexity* of the translations.

For getting some information about the complexity of our compiler, we generated three different series of tests that were conducted with our compiler (`scil1`), the LEGO.NET compiler (`ci11`), and the ahead-of-time compiler of the MONO project (`mono`), all for INTEL's i386 platform with maximum optimisation enabled (`-O3` for the GCC compilers and `-optimize=all` for `mono`). The results are quite interesting. Most of the tests were conducted a 100 times, but for the tests that took longer we resorted to less executions of 20, with a setup similar to the one above. All values given here are the mean values with standard deviations ranging from 0.057 to 0.169.

The first test series consisted of 14 tests. For each test, a C# source file was generated that contained the `Main` method only. In test n that method had 2^{n-1} calls to `System.Console.WriteLine` with an argument increased by 1, eg for $n = 3$:

```
using System;
public static class TestMuch
{
    static void Main ()
    {
        Console.WriteLine (1);
        Console.WriteLine (2);
        Console.WriteLine (3);
        Console.WriteLine (4);
    }
}
```

Unfortunately, `ci11` produced an empty file as output for $n = 14$. The execution times are listed in table 5.1 (in seconds). We can see here, that execution times are all right up to test 10 with 512 lines of code. But after that, execution times dramatically increase for the GCC frontends, much more than those of `mono`. Nevertheless, for all three compilers, the complexity seems to be $O(n^2)$, since in the later tests, doubling of lines of code means four time longer execution times.

In the second test series, we had 19 tests for each compiler similar to the first test series. Instead of calling `System.Console.WriteLine` in each line, we add up an increasing value, eg for $n = 3$:

```
using System;
public static class TestMuch
{
    static void Main ()
    {
        int x = 0;
        x += 1; x += 2; x += 3; x += 4;
        Console.WriteLine (x);
    }
}
```

Test	LOC	scil	cil1	mono
1	1	2.899	2.833	0.946
...
9	256	3.397	3.315	0.985
10	512	3.472	4.600	1.000
11	1024	4.581	10.805	1.078
12	2048	10.935	41.386	1.297
13	4096	40.224	172.531	4.453
14	8192	167.328	—	18.506

Table 5.1: Compiler execution times (test series 1)

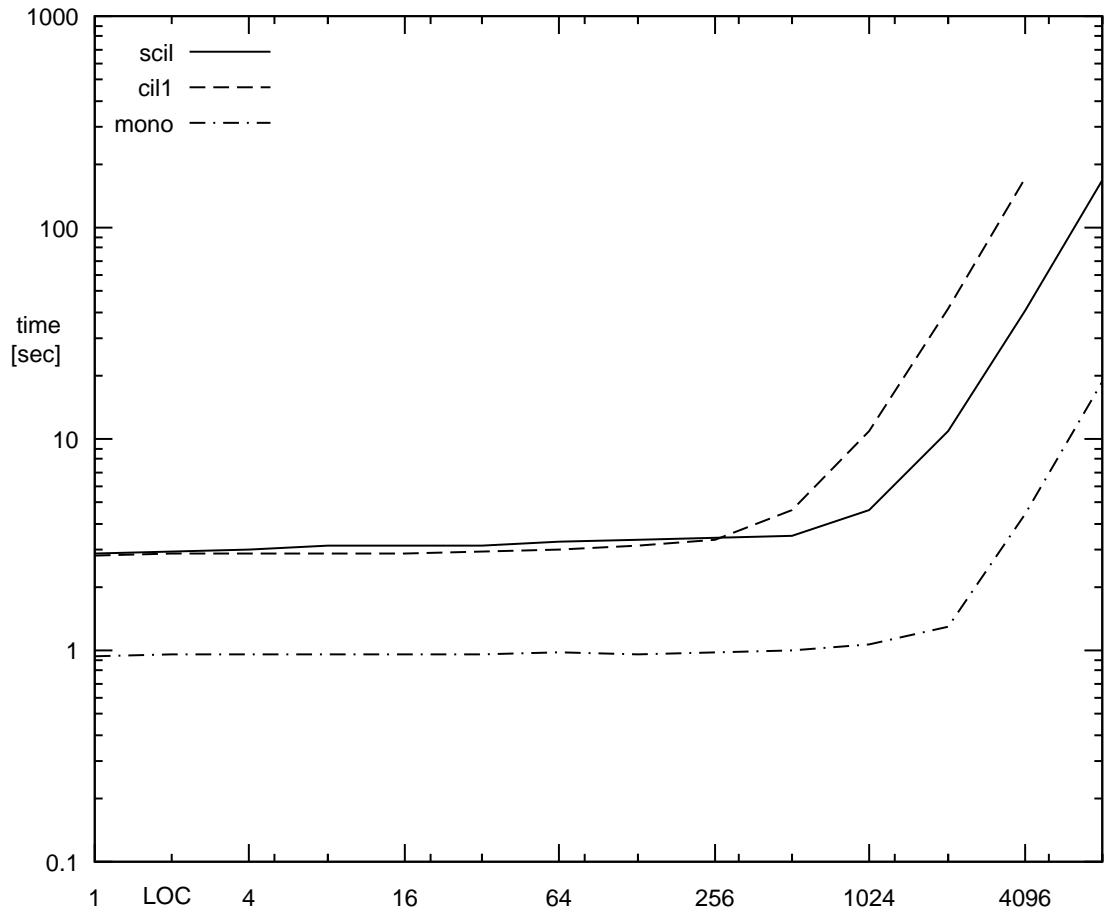


Figure 5.3: Graphical display of compiler execution times (test series 1)

The intent was to force the compilers to optimise that to a single call to `Console.WriteLine` with a *constant* argument. Examination of the produced files showed that the C# compiler did not perform any optimisation but our three test candidates did. Again, `cil1` did produce empty output files for test cases $n \geq 15$. Execution times are presented in table 5.2. Amazingly, the execution times of our compiler are smaller than those of `mono` when you have many lines of code—or bigger assemblies, we might add.

Test	LOC	scil	cil1	mono
1	1	2.903	2.840	1.023
...
10	512	3.094	2.968	0.937
11	1024	3.282	3.047	1.016
12	2048	3.704	3.281	1.047
13	4096	4.422	3.640	1.157
14	8192	6.109	4.406	1.953
15	16384	9.266	—	6.234
16	32768	15.500	—	23.109
17	65536	36.782	—	91.546
18	131072	92.609	—	365.718
19	262144	689.672	—	1461.578

Table 5.2: Compiler execution times (test series 2)

For our final test series, we implemented a poor kind of recursion, in order to not to have all code in one method but lots of methods instead. Now, every test case n consists of a `Main` method and a method `f0`, as well as 2^{n-1} methods `int f1 (int x) to int f2n-1` (`int x`), with f_i returning $f_{i-1}(x) + i$, eg for $n = 3$:

```
using System;
public static class TestMuch
{
    static void Main ()
    {
        Console.WriteLine (f4 (0));
    }

    static int f0(int x) {return x;}
    static int f1 (int x) { return f0 (x) + 1;}
    static int f2 (int x) { return f1 (x) + 2;}
    static int f3 (int x) { return f2 (x) + 3;}
    static int f4 (int x) { return f3 (x) + 4;}
}
```

Now, this test series imposed quite a problem to all compilers; in test case 12 we had to abort the compilation run of `mono` because the system was running out of conventional and virtual memory. From test cases 14 and 16 on, respectively, the other two compilers did not produce any output other than empty files.

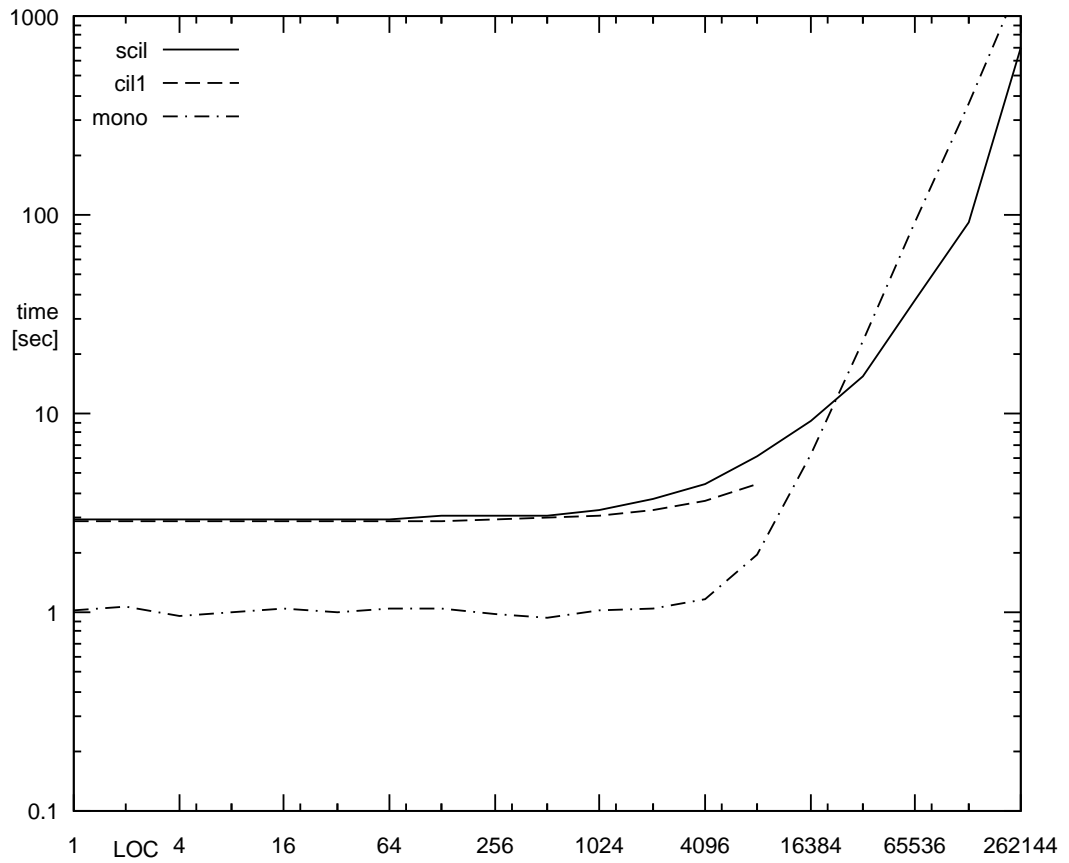


Figure 5.4: Graphical display of compiler execution times (test series 2)

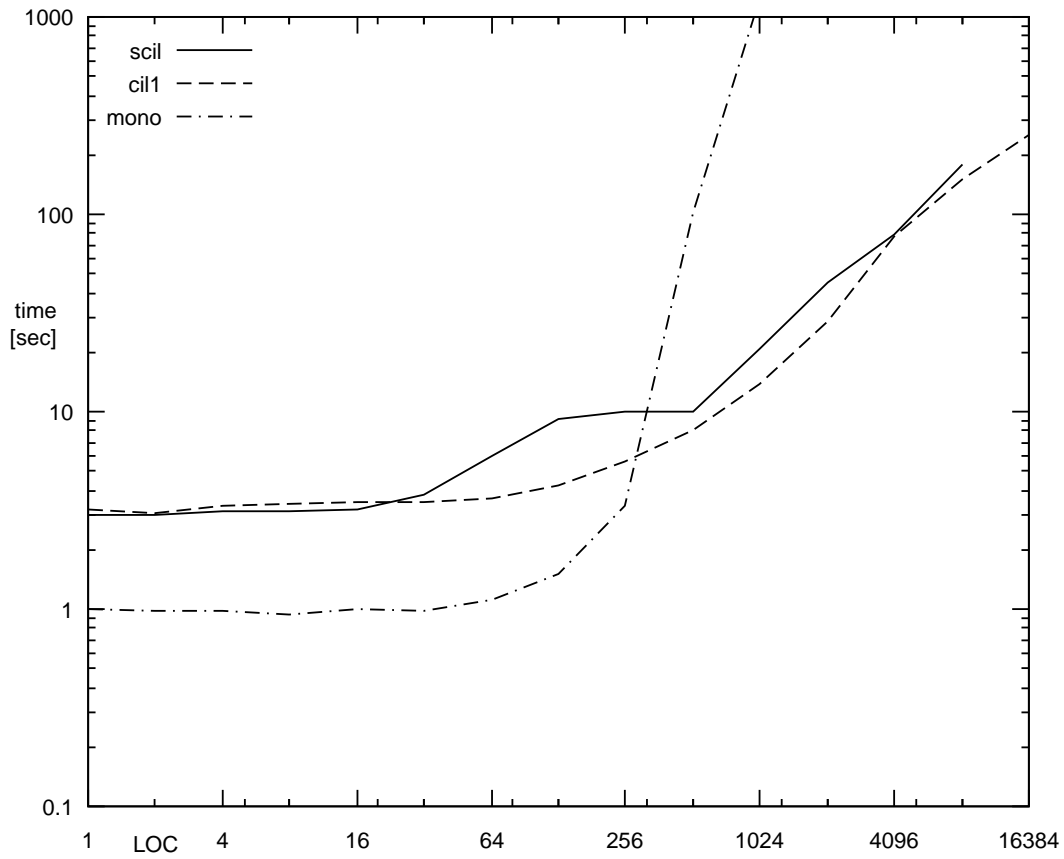


Figure 5.5: Graphical display of compiler execution times (test series 3)

Test	LOC	scil	ci11	mono
1	1	2.968	3.219	1.000
...
5	16	3.218	3.453	1.000
6	32	3.766	3.500	0.985
7	64	6.000	3.656	1.125
8	128	9.234	4.266	1.500
9	256	9.906	5.610	3.360
10	512	10.094	8.125	101.859
11	1024	20.578	13.687	1250.000
12	2048	44.719	28.875	—
13	4096	78.312	76.422	—
14	8192	178.844	151.000	—
15	16384	—	250.063	—

Table 5.3: Compiler execution times (test series 3)

The most important thing we can learn from our performance evaluation is that all tested compilers have their weaknesses. In general, our compiler does perform inside sensible ranges, but it is far from being a perfect tool to be used in a productive application domain.

Chapter 6

Examples

In the last chapter, we were showing how the compiler was implemented in order to translate the extensions we defined in chapter 4. Further, we showed how well the compiler performed in terms of time and lines of code. In contrast, this chapter shall illuminate the *quality* of the translation. For that purpose, we developed some working examples for an embedded system. These examples are thought of to be representative, ie they cover all extensions we implemented along with a thorough analysis of the generated code.

The examples presented in this chapter are all about generating sound on the LEGO MINDSTORM RCX, which is an oversized LEGO brick containing a RENESAS H8/3292 board, a small liquid-crystal display, three input ports for sensors and three output ports for actuators, as well as a simple speaker [PROUDFOOT, 1999].

The H8/3292 is a member of the H8/3297 series [Man H8/3297]. It contains a 16 MHz H8/300 CPU with eight 16-bit general registers and 64-kbyte address space that can be considered a RISC architecture with its 57 instructions most of which execute in two or four states [Man H8/300]. There is no separate address space for peripherals, so we will use our `MemoryAliasAttribute`.

Besides seven I/O ports, a serial communication interface, and an A/D Converter, the board comprises a 16-bit free-running timer, an 8-bit timer module with two distinct channels, and a watchdog timer. In appendix A, we list a complete specification¹ of the series in C#.

6.1 Hardware Access

Our first example shows how to convince the speaker of emitting sounds. The speaker has been connected to channel 0 of the 8-bit timer, such that the state of the compare-match output pin of channel 0 corresponds to the state of the magnet that moves the speaker membrane. In order to produce sounds, we want channel 0 to output a square wave signal, thus moving the membrane. We need to instruct the 8-bit timer to change the state of that pin whenever the timer counter matches the given compare value; in fact, twice as often as the frequency of the sound. We can do so by setting bits 0 and 1 of the timer control/status register (TCSR) of channel 0 [Man H8/3297, § 9.2.4].

Furthermore, we need to tell the timer, *when* to invert the pin. As the name suggests, there is an 8-bit register for that purpose, the time constant register A (TCRA) [Man H8/3297, § 9.2.2]. In every cycle of the RCX's internal clock \emptyset , the value in that register is compared to one of $\emptyset/2$, $\emptyset/8$, $\emptyset/32$, $\emptyset/64$, $\emptyset/256$, or $\emptyset/1024$ depending on bits 0 to 2 of the timer counter register and bit 0 of the serial/timer control register (STCR) [Man H8/3297, § 9.2.2]. Note, that we must invert the pin *twice* for each period, because one period consist of an on and an off state. Hence the timer frequency is

¹We could actually enhance this specification by defining all bits as enum constants. But since enums are not implemented yet...

twice as much as the sound's frequency.

Obviously, we have best granularity when using $\varnothing/2$. \varnothing runs at 16 MHz, $\varnothing/2$ therefore at 8 MHz. But then, $8,000,000/256/2 = 15,625$ Hz will not be playable. Thus, we must sacrifice granularity and use $\varnothing/8$ for lower frequencies, and the other clocks for even lower frequencies. In table 6.1, you can see the frequency ranges for each clock.

Clock	Minimum	Maximum
$\varnothing/2$	15,626	4,000,000
$\varnothing/8$	3,907	1,000,000
$\varnothing/32$	977	250,000
$\varnothing/64$	489	125,000
$\varnothing/256$	123	31,250
$\varnothing/1024$	31	7,812.5

Table 6.1: Frequency Ranges of the RCX's 8-bit Timer

On the other hand, we cannot use $\varnothing/1024$ because we could not play a frequency of eg 131 Hz: $\lfloor 7,813/59 \rfloor = 132$ and $\lfloor 7,813/60 \rfloor = 130$

6.1.1 C# Program

Putting everything together, we get the following methods defined in a class `Lego.Speaker`:

```
public static void Sound (ushort frequency)
{
    Off ();

    if (frequency < 31)
        return;

    H8_3297.Board.Timer8Bit.Channel0.ControlStatusRegister = 0x03;
    H8_3297.Board.Timer8Bit.Channel0.TimerCounter = 0x00;

    if (frequency <= 122)
    {
        H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
            (byte) ( (short) 7813 / frequency);

        H8_3297.Board.SerialTimerControlRegister &= 0xFE;
        H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x0B;
    }
    else if (frequency <= 488)
    {
        H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
            (byte) (31250 / frequency);

        H8_3297.Board.SerialTimerControlRegister |= 0x01;
        H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x0B;
    }
}
```

```
else if (frequency <= 976)
{
    H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
        (byte) (125000 / frequency);

    H8_3297.Board.SerialTimerControlRegister &= 0xFE;
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x0A;
}
else if (frequency <= 3906)
{
    H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
        (byte) (250000 / frequency);

    H8_3297.Board.SerialTimerControlRegister |= 0x01;
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x0A;
}
else if (frequency <= 15625)
{
    H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
        (byte) (1000000 / frequency);

    H8_3297.Board.SerialTimerControlRegister &= 0xFE;
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x09;
}
else
{
    H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
        (byte) (4000000 / frequency);

    H8_3297.Board.SerialTimerControlRegister |= 0x01;
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x09;
}
}

public static void Off ()
{
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0x00;
}
```

6.1.2 Translation Result

Our compiler translates this to the following (shortened and commented) code. Note, that `Speaker.Off` has been inlined and that `r6` contains the frame pointer (FP) and `r7` the stack pointer (SP). The numbers after each instruction display how many cycles each instruction takes, as needed in the performance analysis later on.

```
__ZN3RCX4Lego7Speaker5SoundEt:
* mov.w r6,@-r7      ; 14 save previous FP on stack
* mov.w r7,r6        ; 6 set FP as current SP
  mov.w @(4,r6),r1   ; 18 move function argument to r1
  mov.w #30,r2       ; 12
  cmp.w r2,r1        ; 12
  bgt .L31           ; 12 if frequency > 30
  sub.b r21,r21      ; 6
  mov.b r21,@-56:8   ; 9 TCR = 0
```

```

* jmp    @.L45          ; 14
.L31:
  mov.b  #3,r21        ; 6
  mov.b  r21,@-55:8    ; 9  TCSR = 3
  mov.w  #122,r2       ; 12
  cmp.w  r2,r1         ; 6
  bgt   .L34          ; 12 if frequency > 122
  mov.w  #7813,r0      ; 12
  jsr   @___divhi3    ; 20
  mov.b  r0l,@-54:8    ; 9  TCRA = 7813 / frequency
~ mov.b  @-61:8,r21    ; 9
~ and   #-2,r21       ; 6
~ mov.b  r21,@-61:8    ; 9  STCR &= ~0x01
  mov.b  #11,r21      ; 6
  mov.b  r21,@-56:8    ; 9  TCR = 0x0B
* jmp   @.L45         ; 14
.L34:
  mov.w  #488,r2       ; 12
  cmp.w  r2,r1        ; 6
  bgt   .L36          ; 12 if frequency > 488
  mov.w  #31250,r0     ; 12
  jsr   @___divhi3    ; 20
  mov.b  r0l,@-54:8    ; 9  TCRA = 31250 / frequency
~ mov.b  @-61:8,r21    ; 9
~ or    #1,r21        ; 6
~ mov.b  r21,@-61:8    ; 9  STCR |= 0x01
  mov.b  #11,r21      ; 6
  mov.b  r21,@-56:8    ; 9  TCR = 0x0B
* jmp   @.L45         ; 14
.L36:
  mov.w  #976,r2       ; 12
  cmp.w  r2,r1        ; 6
  bgt   .L38          ; 12 if frequency > 976
  mov.w  r1,r3         ; 6  r2r3 = frequency
  bld   #7,r3h        ; 6
  subx  r2l,r2l       ; 6
  subx  r2h,r2h       ; 6  sign extend r3 into r2
  mov.w  #1,r0        ; 12
  mov.w  #59464,r1     ; 12  r0r1 = 125000
  jsr   @___divsi3    ; 20
  mov.b  r1l,@-54:8    ; 9  TCRA = 125000 / frequency
~ mov.b  @-61:8,r21    ; 9
~ and   #-2,r21       ; 6
~ mov.b  r21,@-61:8    ; 9  STCR &= ~0x01
  mov.b  #10,r21      ; 6
  mov.b  r21,@-56:8    ; 9  TCR = 0x0A
* jmp   @.L45         ; 14
.L38:
  ; ... similar to above ...
  bra   .L45          ; 12
.L40:
  ; ... similar to above ...
  bra   .L45          ; 12
.L42:
.L44:
  mov.w  r1,r3         ; 6
  bld   #7,r3h        ; 6
  subx  r2l,r2l       ; 6

```

```
    subx  r2h,r2h      ; 6  r2r3 = frequency
    mov.w #61,r0      ; 12
    mov.w #2304,r1    ; 12 r0r1 = 4000000
    jsr   @___divsi3  ; 20
    mov.b r1l,@-54:8  ; 9  TCRA = 4000000 / frequency
~   mov.b @-61:8,r2l  ; 9
~   or    #1,r2l      ; 6
~   mov.b r2l,@-61:8  ; 9  STCR |= 0x01
    mov.b #9,r2l     ; 6
    mov.b r2l,@-56:8 ; 9  TCR = 0x0A
.L45: ;
*   mov.w @r7+,r6    ; 14 re-establish FP of caller
    rts             ; 20
```

6.1.3 Qualitative Performance Evaluation

The part before `.L31` is optimal if using frame pointers, ie we do not see how you could do better when implementing this directly in assembly language. In the next section, there is a sub-optimal translation for `STCR &= ~0x01`; instead of using `bclr 0, @-61:8` which takes 18 cycles and 4 bytes length in code, the compiler emitted a sequence of `mov`, `and`, and `mov` instructions which takes 24 cycles and 10 bytes length in code. The same problem occurs after label `.L34` for `STCR |= 0x01` and after `.L36`, `.L38`, `.L40`, and `.L44` (marked with `~`).

We are not quite sure yet, whether this is caused by an unfortunate `tree` constructed by our front end and suitable to inhibit that optimisation, or whether the back end does not perform that optimisation. Inspection of the back end's code revealed that the bit set instructions have been considered, at least.

The only other point for optimisation is the frame pointer. In this example, we need no frame pointer, since we do not push anything on the stack. We could thus omit all lines marked with `*`, and substitute the `jumps` with `rts` instructions. We want to remark that this is actually a problem that stems from the middle end and back end, ie it is not the front end's responsibility to care about the frame pointer.

Altogether, the code just presented has seven paths. For every path, we calculated the overhead our code contains (according to [Man H8/300, Appendix C]). For instance, every path contains the first six instructions and the last three instructions. We found out that—in an ideal solution—the first two and the last but one instructions are unnecessary. Therefore, we have 46 cycles overhead in every path. Table 6.2 shows our results; if we consider the paths uniformly distributed, we can average the execution overhead for that function to around 24.0 %. While we are not quite sure whether you can really say that our overhead is one quarter, we think this gives an impression that the generated code is not too bad, notabene in comparison to an *ideal* solution!

Path	Cycles in our code	Cycles in ideal code	Overhead
1	137	89	53.9 %
2	247	193	28.0 %
3	277	223	24.2 %
4	343	289	18.7 %
5	371	319	16.3 %
6	401	343	16.9 %
7	389	354	9.9 %

Table 6.2: Cycles in every path

6.2 Interrupt Handling

As a simple tone is quite boring we want the RCX to emit more fancy sounds. It should start with a sound at 220 Hz and increase that frequency every millisecond by one until it reaches 2200 Hz. Then, we want it to do the reverse process, to decrease until 220 Hz, and then to start all over again.

Our implementation strategy is simple: We have an infinite loop that sets the frequency and makes the CPU switch to the power-saving sleep mode. Whenever an interrupt occurs, the CPU leaves that mode, processes the respective interrupt handler and continues operation after the sleep instruction. In our case, it jumps back to the start of the loop, hence sets the frequency and switches to the sleep mode again. In order to change the frequency, we need a global variable that our “main thread” reads and that is written to by the timer interrupt handler. Then, we can change the frequency once every millisecond.

We use the H8/3297’s 16-bit timer, assuming, that the internal clock is precise enough for our purposes. This timer has an internal 16-bit counter that is continually increased depending on the clock input signal ($\emptyset/2$, $\emptyset/8$, $\emptyset/32$, or external clock) and compared to the values of output compare registers A and B. When one or both of them match, an output compare match interrupt A or B, resp, is generated. The addresses of the interrupt handlers can only be defined in the ROM’s interrupt vector table. Interestingly, the RCX ROM provides for interrupt handler stubs that save register `r6` and call the actual handler in the RAM, instead of simply setting the actual handler’s address in the interrupt vector table. Therefore, our interrupt handler must be defined as a usual function.

But first, we initialise the timer in `Timer.Handler`. We stop the timer and write the address of our handler to the location the ROM handler stub reads for calling the actual handler.² After that, we instruct the timer module to reset the timer counter whenever it

²Note, that since C# 2.0 delegate objects are implicitly created when methods are used at places where delegates are expected. Therefore, while it may seem that we use the method’s address directly, there is a delegate wrapper around it.

matches the value of output compare register A [Man H8/3297, § 8.2.5]; we select $\emptyset/32$ as input signal for letting the timer run at 500 kHz [Man H8/3297, § 8.2.6]; and set the output compare match register A to 500, thus making the timer generate interrupts at 1 kHz, ie once every millisecond. The generation of appropriate timer interrupts is started and stopped by setting and clearing, resp, bit 3 of the interrupt enable register [Man H8/3297, § 8.2.5].

In our handler, we first need to clear the flag that is set by the timer when the timer counter matches value A, because the timer interrupt is not generated again as long as this flag is set. Then, we check whether we are out of range, and set the frequency.

6.2.1 C# Program

```
using Hardware;
using Renesas;
using Lego;

public class Timer
{
    public static void Init ()
    {
        Stop ();

        Rcx.VectorTable.Timer16Bit.OutputCompareA = HandlerA;

        H8_3297.Board.Timer16Bit.ControlStatusRegister = 0x01;
        H8_3297.Board.Timer16Bit.ControlRegister = 0x02;
        H8_3297.Board.Timer16Bit.CompareMatchA = 500;
    }

    public static void Start ()
    {
        H8_3297.Board.Timer16Bit.InterruptEnableRegister |= 0x08;
    }

    public static void Stop ()
    {
        H8_3297.Board.Timer16Bit.InterruptEnableRegister &= ~0x08;
    }

    private static bool up = true;

    private static void HandlerA ()
    {
        H8_3297.Board.Timer16Bit.ControlStatusRegister &= ~0x08;

        if (MainClass.Frequency > 2200)
            up = false;
        else if (MainClass.Frequency < 220)
            up = true;

        if (up)
            MainClass.Frequency++;
        else
    }
}
```



```

        MainClass.Frequency--;
    }
}

public class MainClass
{
    public static ushort Frequency = 220;

    public static void Main ()
    {
        Timer.Init ();
        Timer.Start ();

        while (true)
        {
            Speaker.Sound (Frequency);
            Cpu.Sleep ();
        }
    }
}

```

6.2.2 Translation Result

Since in the last example we have already seen how hardware access is done, we concentrate on the part that contains new material, showing the manually annotated translation of `Timer.Init`. The method `Timer.Stop` and the setter method of property `H8_3297.Board.Timer16Bit.CompareMatchA` are inlined.

```

__ZN6test_c5Timer4InitEv:
    mov.w r6,@-r7
    mov.w r7,r6                ; establish FP
    subs #2,r7                 ; a local variable
    mov.w #-112,r3
    mov.b @r3,r21
    mov.b r21,@(-2,r6)
    and # -9,r21
    mov.b r21,@r3              ; TIER &= ~0x08
    mov.w #__ZN6test_c5Timer8HandlerAEv,r2
    mov.w r2,@-606:16          ; set Handler
    mov.b #1,r21
    mov.b r21,@-111:8          ; TCSR = 0x01
    add.b r21,r21
    mov.b r21,@-106:8          ; TCR = 0x02
    mov.b #-105,r31
    mov.b @r3,r21
    and #-17,r21
    mov.b r21,@r3              ; TOCCR &= ~0x10
    mov.w #500,r2
    mov.w r2,@-108:16          ; TOCRA = 500
    adds #2,r7
    mov.w @r7+,r6              ; re-establish FP
    rts

```

We want to point out that, if we attach `InterruptHandlerAttribute` to `Timer.HandlerA`, the method is translated to (shortened)

```
.global __ZN6test_c5Timer8HandlerAEv
```

```
__ZN6test_c5Timer8HandlerAEv:
    mov.w r6,@-r7
    mov.w r7,r6
    mov.w r2,@-r7
    mov.w r3,@-r7
    mov.b @-111:8,r21
    and    #-9,r21
    mov.b r21,@-111:8
    mov.w @__ZN6test_c9MainClass9FrequencyE:16,r3
    mov.w #2200,r2
    cmp.w r2,r3
    ; ...
    mov.w @r7+,r3
    mov.w @r7+,r2
    mov.w @r7+,r6
    rte
```

The only thing to note here is that `rte` is generated and that all used registers (`r2` and `r3`) are saved on the stack on entry and restored on exit. Nevertheless, as mentioned above, we cannot (or need not) use `InterruptHandlerAttribute` here.

6.2.3 Qualitative Performance Evaluation

As you can see, the code for setting the handler is minimal; its address is moved into `r2` and from there to memory cell `0xFDA2`.

However, we can see another problem here. The GCC decides to inline `Timer.Stop` but interestingly enough there is a difference between the inlined and non-inlined versions:

```
__ZN6test_c5Timer4StopEv:
    mov.w r6,@-r7
    mov.w r7,r6
    subs #2,r7
    mov.w #-112,r3
    mov.b @r3,r21
    and    #-9,r21
    mov.b r21,@r3
    adds #2,r7
    mov.w @r7+,r6
    rts
```

When inlining, the GCC adds the absolutely unnecessary `mov.b r21,@(-2,r6)`. More strangely, there are reservations for a local variable in *both* versions. This might indicate an incomplete optimisation process but we cannot back that up. Further, the translations of `or` and `and` are even less efficient than in our first example. We cannot see why it is like that.

Our code calls a runtime method `Hardware.Cpu.Sleep`, which we defined in section 4.4.1. We implemented that runtime method in assembly code:

```
__ZN5CliHw8Hardware3Cpu5SleepEv:
    bclr #7, @0xC4:8
    sleep
    rts
```

We think, however, that back ends should be extended to directly inline such a `sleep` instruction instead of calling a method. But since performance can never be crucial at that point this is to be considered a minor issue.

6.3 Scheduling

Finally, we want to present a simple scheduler for two threads.

6.3.1 C# Program

We re-use the last example's `Timer` class but need to slightly modify the `Init` and `HandlerA` methods; we further add a static delegate field `Scheduler`. `HandlerA` has a different signature now: it gets the current stack pointer as a parameter and returns the new stack pointer to be used after it has returned. Obviously, we cannot directly hook in `HandlerA` to the RCX's timer interrupt handler stub. Instead, in `Timer.Init`, we put its address in the static delegate field `Cpu.Scheduler` such that it is called whenever `Cpu.InvokeScheduler` is called. We then connect `Cpu.InvokeScheduler` to the RCX's timer interrupt handler stub.

```
public class Timer
{
    public static Cpu.SchedulerDelegate Scheduler;

    public static void Init ()
    {
        Stop ();

        Cpu.Scheduler = HandlerA;
        Rcx.VectorTable.Timer16Bit.OutputCompareA = Cpu.InvokeScheduler;

        H8_3297.Board.Timer16Bit.ControlStatusRegister = 0x01;
        H8_3297.Board.Timer16Bit.ControlRegister = 0x02;
        H8_3297.Board.Timer16Bit.CompareMatchA = 500;
    }
    public static Cpu.Stack HandlerA (Cpu.Stack stack)
    {
        H8_3297.Board.Timer16Bit.ControlStatusRegister &= ~0x08;
        return Scheduler (stack);
    }
}
```

Because we do not have any memory management at the moment, we use a simple value type `Thread` for implementing threads.³ All this type contains is a field for storing the stack pointer. We further declare two static fields that contain our two threads, `Thread1` and `Thread2`, as well as two static methods containing the code for each thread, `ThreadCode1` and `ThreadCode2`. They do nothing else but setting the frequency and then

³If we had a class here we had to create objects on the heap. Using a value type, static variables can be declared the space for which is part of the code, ie statically reserved.

go to sleep in an infinite loop. We thus have a simple program changing the sound once a second. In the static constructor, we initialise the stack of thread 2 to a value we can assume not to be used by our program. We need not initialise thread 1 as it will be our main.

```
struct Thread
{
    public Cpu.Stack stack;
    public static Thread Thread1;
    public static Thread Thread2;

    static Thread ()
    {
        Thread2.stack = Cpu.InitStack (ThreadCode2, 0xE000);
    }
    public static void ThreadCode1 ()
    {
        while (true)
        {
            Speaker.Sound (880);
            Cpu.Sleep ();
        }
    }
    static void ThreadCode2 ()
    {
        while (true)
        {
            Speaker.Sound (440);
            Cpu.Sleep ();
        }
    }
}
```

Our scheduler is as simple (read primitive) as can be. It is called every millisecond; since we want a timeslice of one second, we need to wait for the thousandth call. We then switch the flag and choose the thread accordingly. As you might guess, any scheduling algorithm could be implemented in that method.

```
class Scheduler
{
    static bool flag = false;
    static ushort count = 0;

    public static Cpu.Stack DetermineNext (Cpu.Stack stack)
    {
        if (count++ == 1000)
        {
            count = 0;
            flag = !flag;

            if (flag)
            {
                Thread.Thread1.stack = stack;
                return Thread.Thread2.stack;
            }
            else
        }
    }
}
```

```

    {
        Thread.Thread2.stack = stack;
        return Thread.Thread1.stack;
    }
}
return stack;
}
}

```

Lastly, we have to initialise the timer, connect the scheduler to the timer, and start the timer. As said above, thread 1 is just the continuation of our Main, hence we need to call its code here.

```

public class MainClass
{
    public static void Main ()
    {
        Timer.Init ();
        Timer.Scheduler = Scheduler.DetermineNext;
        Timer.Start ();
        Thread.ThreadCode1 ();
    }
}

```

6.3.2 Runtime Methods

There is nothing new in that example except for the runtime methods `Hardware.Cpu.InvokeScheduler` and `Hardware.Cpu.InitStack` the implementations of which were written in assembly code:

```

__ZN5CliHw8Hardware3Cpu15InvokeSchedulerEv:
    ; save context, r6 has been saved by ROM
    push r0
    push r1
    push r2
    push r3
    push r4
    push r5

    ; pass current stack pointer
    push r7

    ; pass location for new stack pointer
    push r7

    ; call scheduler method
    mov.w @__ZN5CliHw8Hardware3Cpu9SchedulerE:16, r2
    jsr @r2

    ; pop new stack pointer
    pop r7

    ; adjust stack
    adds #2, r7

```

```
    ; restore context, r6 will be restored by ROM
    pop r5
    pop r4
    pop r3
    pop r2
    pop r1
    pop r0

    rts

__ZN5CliHw8Hardware3Cpu9InitStackEPvt:

    ; move stack pointer in r1
    mov.w @(6,r7), r1

    ; prepare stack
    ; push pc
    mov.w @(4,r7), r2
    mov.w r2, @r1

    ; push ccr
    sub.w r2, r2
    mov.w r2, @-r1

    ; push r6
    mov.w r2, @-r1

    ; push timer interrupt return address
    mov.w #0x04d4, r2
    mov.w r2, @-r1

    ; reserve space for registers r0 - r5
    add.b #0xF4, r1l
    addx #0xFF, r1h

    ; push stack pointer
    mov.w r1, @-r1

    ; return stack pointer
    mov.w @(2,r7), r2
    mov.w r1, @r2

    rts
```

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have shown that standards describing general-purpose virtual machines can be extended quite easily for the programming of specific hardware.

Because of the strong similarities, we presented a general approach for accessing memory-mapped and port-based hardware. Additional extensions render possible the implementation of immediate interrupt handlers and context switches from *inside* the virtual machine, ie without the need for native plug-ins written in another programming system.

With simple but representative examples we have made clear how these extensions can be used for the tasks we intended. By generating considerably efficient assembly code, our proof-of-concept, ahead-of-time compiler demonstrates, that programs for virtual machines can appropriately be written even for machine-oriented programming. In addition, the way our compiler works results in minimal-sized images because it includes only those methods of standard and custom libraries that are actually used by the program.

The opportunity to separate the specification of hardware locations from their use in applications allows more transparency for users of that code; compilers could be enhanced to disallow certain parts of a program to use memory alias attributes on fields, thus making it possible to restrict the specification of hardware locations to trusted OEM libraries. Also, analysis tools can effortlessly find out, which hardware locations are accessed, eventually enabling programmers do find bugs faster and giving users more insight about the programs they use. Even the code itself becomes more readable and hence better maintainable because programmers are forced to use descriptive names in lieu of cryptic numbers then.

7.2 Outlook

With respect to our front end and the GCC in general, we suggest various optimisations. For example, the concept of interrupt handlers should be generalised such that all back ends implement the same attribute (possibly allowing both attributes for *all* back ends). Additionally, it would be nice to have either special tree nodes or built-in functions for port based I/O operations and the sleep instruction(s), to avoid overhead by linking the generated object files to special runtime files. Another—much harder—task would be an optimised integration of context switches and context initialisation.

As another extension, we deem it necessary to provide for attributes that allow programmers to specify certain segments that variables and code should be placed in. Sensible usage includes the definition of ROM code and code that shall run in memory areas that are faster accessible than others. Moreover, there should be some mechanism that allows the implementation of memory management modules.

Appendix A

Classes for the Renesas H8/3297 series

A.0.1 Board.cs

```
using Hardware;

namespace Renesas
{
    [StaticDelegate]
    public delegate void InterruptHandler ();

    public struct H8_3297
    {
        [MemoryAlias (0x0000)]
        public static H8_3297 Board;

        [MemoryAlias (0x0000)]
        public VectorTable VectorTable;
        /* If programming against ROM use this line: */
        /* public readonly VectorTable VectorTable; */

        [MemoryAlias (0xFF90)]
        public Timer16Bit Timer16Bit;

        [MemoryAlias (0xFFA8)]
        public WatchdogTimer WatchdogTimer;

        [MemoryAlias (0xFFB0)]
        public PortIpcr Port1;

        [MemoryAlias (0xFFB1)]
        public PortIpcr Port2;

        [MemoryAlias (0xFFB4)]
        public PortIpcr3 Port3;

        [MemoryAlias (0xFFB5)]
        public Port Port4;

        [MemoryAlias (0xFFB8)]
        public Port Port5;

        [MemoryAlias (0xFFB9)]
        public Port Port6;

        [MemoryAlias (0xFFBE)]
        public sbyte Port7;

        [MemoryAlias (0xFFC2)]
        public sbyte WaitStateControlRegister;

        [MemoryAlias (0xFFC3)]
        public sbyte SerialTimerControlRegister;

        [MemoryAlias (0xFFC4)]
        public sbyte SystemControlRegister;

        [MemoryAlias (0xFFC5)]
        public sbyte ModeControlRegister;
    }
}
```

```

    [MemoryAlias (0xFFC6)]
    public sbyte IrqSenseControlRegister;

    [MemoryAlias (0xFFC7)]
    public sbyte IrqEnableRegister;

    [MemoryAlias (0xFFC8)]
    public Timer8Bit Timer8Bit;

    [MemoryAlias (0xFFD8)]
    public SerialCommunication SerialCommunication;

    [MemoryAlias (0xFFE0)]
    public ADConverter ADConverter;
}

public struct VectorTable
{
    [MemoryAlias (0x00)]
    public InterruptHandler AfterReset;

    [MemoryAlias (0x06)]
    public InterruptHandler NonMaskableInterrupt;

    [MemoryAlias (0x08)]
    public InterruptHandler Irq0;

    [MemoryAlias (0x0A)]
    public InterruptHandler Irq1;

    [MemoryAlias (0x0C)]
    public InterruptHandler Irq2;

    [MemoryAlias (0x18)]
    public Timer16Bit.VectorTable Timer16Bit;

    [MemoryAlias (0x26)]
    public Timer8Bit.VectorTable Timer8Bit;

    [MemoryAlias (0x36)]
    public SerialCommunication.VectorTable SerialCommunication;

    [MemoryAlias (0x46)]
    public ADConverter.VectorTable ADConverter;

    [MemoryAlias (0x48)]
    public WatchdogTimer.VectorTable WatchdogTimer;
}
}

```

A.0.2 ADConverter.cs

```

using Hardware;
using Hardware.BigEndian;

namespace Renesas
{
    public class ADConverter

```

```
{
    [MemoryAlias (0x00)]
    public TwoBytes DataRegisterA;

    [MemoryAlias (0x02)]
    public TwoBytes DataRegisterB;

    [MemoryAlias (0x04)]
    public TwoBytes DataRegisterC;

    [MemoryAlias (0x06)]
    public TwoBytes DataRegisterD;

    [MemoryAlias (0x08)]
    public sbyte ControlStatusRegister;

    [MemoryAlias (0x09)]
    public sbyte ControlRegister;

    public class VectorTable
    {
        [MemoryAlias (0x00)]
        public InterruptHandler ConversionEnd;
    }
}
```

A.0.3 Port.cs

```
using Hardware;

namespace Renesas
{
    public struct PortIpcr
    {
        [MemoryAlias (0x00)]
        public sbyte InputPullUpControlRegister;

        [MemoryAlias (0x04)]
        public Port port;
    }

    public struct PortIpcr3
    {
        [MemoryAlias (0x00)]
        public sbyte InputPullUpControlRegister;

        [MemoryAlias (0x06)]
        public Port port;
    }

    public struct Port
    {
        [MemoryAlias (0x00)]
        public sbyte DataDirectionRegister;

        [MemoryAlias (0x02)]
        public sbyte DataRegister;
    }
}
```

```
    }  
}
```

A.0.4 SerialCommunication.cs

```
using Hardware;  
  
namespace Renesas  
{  
    public struct SerialCommunication  
    {  
        [MemoryAlias (0x00)]  
        public sbyte SerialModeRegister;  
  
        [MemoryAlias (0x01)]  
        public byte BitRateRegister;  
  
        [MemoryAlias (0x02)]  
        public sbyte SerialControlRegister;  
  
        [MemoryAlias (0x03)]  
        public sbyte TransmitDataRegister;  
  
        [MemoryAlias (0x04)]  
        public sbyte SerialStatusRegister;  
  
        [MemoryAlias (0x05)]  
        public sbyte ReceiveDataRegister;  
  
        public struct VectorTable  
        {  
            [MemoryAlias (0x00)]  
            public InterruptHandler ReceiveError;  
  
            [MemoryAlias (0x02)]  
            public InterruptHandler ReceiveEnd;  
  
            [MemoryAlias (0x04)]  
            public InterruptHandler TdrEmpty;  
  
            [MemoryAlias (0x06)]  
            public InterruptHandler TsrEmpty;  
        }  
    }  
}
```

A.0.5 Timer8Bit.cs

```
using Hardware;  
using Hardware.BigEndian;  
  
namespace Renesas  
{  
    public struct Timer8Bit  
    {  
        [MemoryAlias (0x00)]  
        public Channel Channel0;  
    }  
}
```

```
[MemoryAlias (0x08)]
public Channel Channel1;

public struct VectorTable
{
    [MemoryAlias (0x00)]
    public Channel.VectorTable Channel0;

    [MemoryAlias (0x06)]
    public Channel.VectorTable Channel1;
}

public struct Channel
{
    [MemoryAlias (0x00)]
    public sbyte ControlRegister;

    [MemoryAlias (0x01)]
    public sbyte ControlStatusRegister;

    [MemoryAlias (0x02)]
    public byte ConstantRegisterA;

    [MemoryAlias (0x03)]
    public byte ConstantRegisterB;

    [MemoryAlias (0x04)]
    public byte Counter;

    public struct VectorTable
    {
        [MemoryAlias (0x00)]
        public InterruptHandler CompareMatchA;

        [MemoryAlias (0x02)]
        public InterruptHandler CompareMatchB;

        [MemoryAlias (0x04)]
        public InterruptHandler Overflow;
    }
}
}
```

A.0.6 Timer16Bit.cs

```
using Hardware;
using Hardware.BigEndian;

namespace Renesas
{
    public struct Timer16Bit
    {
        [MemoryAlias (0x00)]
        public sbyte InterruptEnableRegister;

        [MemoryAlias (0x01)]
```

```

public sbyte ControlStatusRegister;

[MemoryAlias (0x02)]
public ushort FreeRunningCounter;

[MemoryAlias (0x04)]
public ushort OutputCompareRegister;

[MemoryAlias (0x06)]
public sbyte ControlRegister;

[MemoryAlias (0x07)]
public sbyte OutputCompareControlRegister;

[MemoryAlias (0x08)]
public ushort InputCaptureRegisterA;

[MemoryAlias (0x0A)]
public ushort InputCaptureRegisterB;

[MemoryAlias (0x0C)]
public ushort InputCaptureRegisterC;

[MemoryAlias (0x0E)]
public ushort InputCaptureRegisterD;

public struct VectorTable
{
    [MemoryAlias (0x00)]
    public InterruptHandler InputCaptureA;

    [MemoryAlias (0x02)]
    public InterruptHandler InputCaptureB;

    [MemoryAlias (0x04)]
    public InterruptHandler InputCaptureC;

    [MemoryAlias (0x06)]
    public InterruptHandler InputCaptureD;

    [MemoryAlias (0x08)]
    public InterruptHandler OutputCompareA;

    [MemoryAlias (0x0A)]
    public InterruptHandler OutputCompareB;

    [MemoryAlias (0x0C)]
    public InterruptHandler Overflow;
}

public ushort CompareMatchA
{
    set
    {
        H8_3297.Board.Timer16Bit.OutputCompareControlRegister &= ~0x10;
        H8_3297.Board.Timer16Bit.OutputCompareRegister = value;
    }
}

```

```
        public ushort CompareMatchB
        {
            set
            {
                H8_3297.Board.Timer16Bit.OutputCompareControlRegister |= 0x10;
                H8_3297.Board.Timer16Bit.OutputCompareRegister = value;
            }
        }
    }
}
```

A.0.7 WatchdogTimer.cs

```
using Hardware;

namespace Renesas
{
    public struct WatchdogTimer
    {
#pragma warning disable 0649
        [MemoryAlias (0x00)]
        private sbyte controlStatusRegister;

        [MemoryAlias (0x01)]
        private byte counter;
#pragma warning restore 0649

        [MemoryAlias (0x00)]
        private ushort wordAccess;

        public byte Counter
        {
            get { return counter; }
            set
            {
                wordAccess = (ushort) (0x5A00 | (ushort) value);
            }
        }

        public sbyte ControlStatusRegister
        {
            get { return controlStatusRegister; }
            set
            {
                wordAccess = (ushort) (0xA500 | (ushort) value);
            }
        }

        public struct VectorTable
        {
            [MemoryAlias (0x00)]
            public InterruptHandler Overflow;
        }
    }
}
```


Bibliography

- [.NETCF, 2006] MICROSOFT CORPORATION, 2006: *Comparisons with the .NET Framework*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_evtuv/html/etcondifferenceswithnetframework.asp (September 23, 2006)
- [AICAS a, 2006] AICAS GMBH, 2006: *Introduction*. <http://www.aicas.com/jamaica/doc/html/intro.html> (September 23, 2006)
- [AICAS b, 2006] AICAS GMBH, 2006: *Introduction*. <http://www.aicas.com/jamaica/doc/html/intro-mf.html> (September 23, 2006)
- [BACON, 2005] David F. BACON, Perry CHENG, David GROVE, Michael HIND, V. T. RAJAN, Eran YAHAV, Matthias HAUSWIRTH, Christoph M. KIRSCH, Daniel SPOONHOWER, Martin T. VECHEV, 2005. *High-level real-time programming in Java*. in *EMSOFT* [WOLF, 2005], p 68-78. ISBN 1-59593-091-4. <http://doi.acm.org/10.1145/1086228.1086242>
- [BARR, 2006] Michael BARR, Anthony MASSA, 2006. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media. ISBN 0596009836.
- [BELLINI, 2000] Pierfrancesco BELLINI, R. MATTONLINI, Paolo NESI, 2000: *Temporal logics for real-time system specification*. in *ACM Comput. Surv.* vol. 32 no. 1, p 12-42. <http://doi.acm.org/10.1145/349194.349197>
- [BOLLOW, 2006] Friedrich BOLLOW, 2006. *C und C++ fr Embedded Systems*. Mitp-Verlag. ISBN 3826616189.
- [BROSGOL, 2001] Benjamin M. BROSGOL, Brian DOBBING, 2001. *Real-time convergence of Ada and Java*. in *SIGAda*, p 11-26.
- [BUSCHMANN, 1996] Frank BUSCHMANN, Regine MEUNIER, Hans ROHNERT, Peter SOMMERLAD, Michael STAL, 1996. *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley & Sons Ltd, West Sussex, England. ISBN 0-471-95869-7.
- [CMMI, 2006] CARNEGIE MELLON SOFTWARE ENGINEERING INSTITUTE, 2006: *CMMI Main Page*. <http://www.sei.cmu.edu/cmmi/> (September 26, 2006)
- [CSABI, 2001] CODESOURCERY LLC, March 21, 2001: *C++ ABI Summary*. <http://www.codesourcery.com/cxx-abi/> (May 20, 2006)
- [CHIAO, 2002] Hsin-Ta CHIAO, Scott Hsu-Jing KAO, Yue-Shan CHANG, Shen-Tzay HUANG, Shyan-Ming YUAN, 2002: *Experience in Building a Real-Time Extension Library for Java*. in *J. Inf. Sci. Eng.* vol. 18 no. 6, p 905-927. http://www.iis.sinica.edu.tw/JISE/2002/200211_04.html

- [CORSARO, 2002] Angelo CORSARO, Douglas C. SCHMIDT, 2002. *The Design and Performance of the jRate Real-Time Java Implementation*. in *CoopIS/DOA/ODBASE* [MEERSMAN, 2002], p 900-921. ISBN 3-540-00106-9. <http://link.springer.de/link/service/series/0558/bibs/2519/25190900.htm>
- [ECMA 334] ECMA standard 334: *C# Language Specification*, 3rd edition, 2005. Note: ISO/IEC approved 2nd edition as ISO/IEC 23270. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>
- [ECMA 335] ECMA standard 335: *Common Language Infrastructure (CLI)*, 3rd edition, 2005. Note: ISO/IEC approved 2nd edition as ISO/IEC 23271. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>
- [ELLIS, 2002] Carla Schlatter ELLIS (ed), 2002. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*. USENIX. ISBN 1-880446-00-6.
- [GCC a, 2006] GNU PROJECT – FREE SOFTWARE FOUNDATION (FSF): *GCC Home Page*. <http://gcc.gnu.org> (February 14, 2006)
- [GCC b, 2006] GNU PROJECT – FREE SOFTWARE FOUNDATION (FSF): *Status of Supported Architectures from Maintainers' Point of View*. <http://gcc.gnu.org/backends.html> (February 14, 2006)
- [GCC c, 2006] GNU PROJECT – FREE SOFTWARE FOUNDATION (FSF): *GCC Front Ends*. <http://gcc.gnu.org/frontends.html> (September 19, 2006)
- [GCCc] GNU PROJECT – FREE SOFTWARE FOUNDATION (FSF): *Standard C++ Library ABI*. <http://gcc.gnu.org/onlinedocs/libstdc++/abi.html> (May 20, 2006)
- [GOLM, 2002] Michael GOLM, Meik FELSER, Christian WAWERSICH, Jürgen KLEINÖDER, 2002. *The JX Operating System*. in *USENIX Annual Technical Conference, General Track* [ELLIS, 2002], p 45-58. ISBN 1-880446-00-6. <http://www.usenix.org/publications/library/proceedings/usenix02/golm.html>
- [HALANG, 1991] Wolfgang A. HALANG, Alexander D. STOYENKO, 1991. *Constructing Predictable Real-time Systems*. Kluwer Academic Publishers, Norwell, Massachusetts. ISBN 0-7923-9202-7.
- [HANSSON, 2005] Hans HANSSON, Mikael NOLIN, Thomas NOLTE, 2005. *Real-Time Systems*. in *The Industrial Information Technology Handbook* [ZURAWSKI, 2005], p 81-1–81-28. ISBN 0-8493-1985-4.
- [HARDIN, 2005] ∞ David S. HARDIN, 2005: *White Paper: aJile Systems: Low-Power Direct-Execution Java Microprocessors for Real-Time and Networked Embedded Applications*. aJile Systems, Inc.. <http://www.ajile.com/downloads/aJile-white-paper.pdf>
- [HSIEH, 2001] Harry HSIEH, Felice BALARIN, Alberto SANGIOVANNI-VINCENTELLI, 2001. *Synchronous Equivalence: Formal Methods for Embedded Systems*. Kluwer Academic Publishers, Norwell, Massachusetts. ISBN 0-7923-7262-X.
- [ISO/IEC 8652:1995] ISO/IEC standard 8652:1995: *Ada Reference Manual*2001. <http://www.ada-auth.org/arm-files/RM.PDF>

-
- [ISORC, 2006] Martin VON LÖWIS, Andreas RASCHE, 2006. *Towards a Real-Time Implementation of the ECMA Common Language Infrastructure*. in *ISORC* [ISORC, 2006], p 125-132. ISBN 0-7695-2561-X. <http://doi.ieeecomputersociety.org/10.1109/ISORC.2006.72>
- [ISORC, 2006] *Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006), 24-26 April 2006, Gyeongju, Korea*. IEEE Computer Society. ISBN 0-7695-2561-X.
- [JME, 2006] SUN MICROSYSTEMS, INC.: *Java*. <http://java.sun.com/javame/index.jsp> (September 15, 2006)
- [JNI, 2003] SUN MICROSYSTEMS, INC., 2003: *JNI - Java Native Interface*. <http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html> (September 23, 2006)
- [JSERT, 2006] SUN MICROSYSTEMS, INC., 2006: *Java SE Real-Time*. <http://java.sun.com/javase/technologies/realtime.jsp> (September 23, 2006)
- [JVMRTS, 2001] Fridtjof SIEBERT, Andy WALTER, 2001. *Deterministic Execution of Java's Primitive Bytecode Operations*. in *Java Virtual Machine Research and Technology Symposium* [JVMRTS, 2001], p 141-152. ISBN 1-880446-11-1. <http://www.usenix.org/publications/library/proceedings/jvm01/siebert.html>
- [JVMRTS, 2001] *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX. ISBN 1-880446-11-1.
- [JXhist, 2006] JX LIMITED: *History*. <http://www.jxos.org/history.html> (August 4, 2006)
- [Jbed, 2006] ESMERTEC AG: *esmertec : wireless software solutions for mass market mobile multimedia phones and embedded devices*. http://www.esmertec.com/solutions/mobile_multimedia/index.shtml (September 15, 2006)
- [Komodo, 2006] UNIVERSITY OF AUGSBURG: *Komodo—Java-microcontroller*. <http://ipr.ira.uka.de/komodo/komodoEng.html> (September 15, 2006)
- [LAVAGNO, 2005] Luciano LAVAGNO, Claudio PASSERONE, 2005. *Design of Embedded Systems*. in *The Industrial Information Technology Handbook* [ZURAWSKI, 2005], p 1-14. ISBN 0-8493-1985-4.
- [LEE, 2002] Edward A. LEE, 2002: *Embedded Systems* in *Advances in Computers* vol. 56.
- [LEIF, 2003] Robert C. LEIF, Ricky E. SWARD (eds), 2003. *Proceedings of the 2003 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies 2003, San Diego, CA, USA, December 7-11, 2003*. ACM. ISBN 1-58113-476-2.
- [LIU, 2000] Jane W. S. LIU, 2000. *Real-time Systems*. Prentice Hall, Inc, Upper Saddle River, New Jersey. ISBN 0-13-099651-3.
- [LUTZ, 2003] Michael H. LUTZ, Phillip A. LAPLANTE, 2003: *IEEE Software: Real-Time Systems - C# and the .NET Framework: Ready for Real Time?* in *IEEE Distributed Systems Online* vol. 4 no. 2, p 74-80. <http://dsonline.computer.org/0302/f/sp1lap.htm>
- [Man H8/300] RENESAS, December 1989: *H8/300 Programming Manual*.

- [Man H8/3297] RENESAS, April 1, 2003: *Renesas Single-Chip Microcomputer H8/3297 Series Hardware Manual*.
- [MEERSMAN, 2002] Robert MEERSMAN, Zahir TARI (eds), 2002. *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings*. Springer. ISBN 3-540-00106-9.
- [Mono, 2006] THE MONO PROJECT: *Main Page*. http://www.mono-project.com/Main_Page (August 4, 2006)
- [NIST 1999] REQUIREMENTS GROUP FOR REAL-TIME EXTENSIONS FOR THE JAVA PLATFORM ∞ Lisa CARNAHAN, Marcus RUARK (eds), 1999: *Requirements For Real-time Extensions For the Java Platform*. National Institute for Standards and Technology. <http://www.nist.gov/itl/div897/ctg/real-time/rtj-final-draft.pdf>
- [NILSEN, 2005] ∞ Kelvin NILSEN, 2005: *White Paper: PERC VM—Differentiating Features and Value-Benefits*. Aonix North America, Inc..
- [OSMG, 2006] OPERATING SYSTEMS AND MIDDLEWARE GROUP AT HPI: *Welcome!*. <http://www.dcl.hpi.uni-potsdam.de>
- [PINHO, 1999] Lus Miguel PINHO, Francisco VASQUES, December 1999: *To Ada or not To Ada: Adaing vs. Javaing in Real-Time Systems* in *ACM Ada Letters* vol. XIX no. 4, p 37-43.
- [POTRATZ, 2003] Eric POTRATZ, 2003. *A practical comparison between Java and Ada in implementing a real-time embedded system*. in *SIGAda* [LEIF, 2003], p 71-83. ISBN 1-58113-476-2. <http://doi.acm.org/10.1145/958420.958431>
- [PROUDFOOT, 1999] STANFORD UNIVERSITY ∞ Kekoa PROUDFOOT, 1999: *RCX Internals*. <http://graphics.stanford.edu/~kekoa/rcx> (August 12, 2006)
- [RTAS, 2002] *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002), 24-27 September 2002, San Jose, CA, USA*. IEEE Computer Society. ISBN 0-7695-1739-0.
- [RTCSA, 1999] Fridtjof SIEBERT, 1999. *Hard Real-Time Garbage-Collection in the Jamaica Virtual Machine*. in *RTCSA* [RTCSA, 1999], p 96-102. ISBN 0-7695-0306-3. <http://csdl.computer.org/comp/proceedings/rtcsa/1999/0306/00/03060096abs.htm>
- [RTCSA, 1999] *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*. IEEE Computer Society. ISBN 0-7695-0306-3.
- [RTJCom, 2006] RTJ COMPUTING PTY. LTD., 2006: *The simple Real Time Java*. <http://www.rtjcom.com/main.php?p=home> (September 23, 2006)
- [RTJS, 2000] THE REAL-TIME FOR JAVA EXPERT GROUP ∞ Greg BOLLELLA, Ben BROSGOL, Peter DIBBLE, Steve FURR, James GOSLING, David HARDIN, Mark TURNBULL, Rudy BELLIARDI, 2000: *The Real-Time Specification for Java*. ISBN 0-201-70323-8. <http://www.rtfj.org>
- [RITCHIE, 2006] ∞ Brian RITCHIE, 2006: <http://www.dotnetpowered.com/languages.aspx>. <http://gcc.gnu.org/frontends.html> (September 19, 2006)

- [SCHNEIDER, 2000] Steve SCHNEIDER, 2000. *Concurrent and Real-time Systems – The CSP approach*. John Wiley & Sons, Ltd, Chichester. ISBN 0-471-62373-3.
- [SHUMATE, 1992] Kenneth C. SHUMATE, Marilyn M. KELLER, 1992. *Software Specification and Design – A Disciplined Approach for Real-time Systems*. John Wiley & Sons, Inc. ISBN 0-471-53296-7.
- [STALLMAN, 2005] ∞ Richard M. STALLMAN, ET. AL., 2005: *GCC Internals*. Free Software Foundation. <http://gcc.gnu.org/onlinedocs/gccint.pdf>
- [STANKOVIC, 1998] John A. STANKOVIC, Marco SPURI, Krithi RAMAMRITHAM, Giorgio C. BUTTAZZO, 1998. *Deadline Scheduling for Real-time Systems*. Kluwer Academic Publishers, Norwell, Massachusetts. ISBN 0-7923-8269-2.
- [STROUSTRUP, 2000] Bjarne STROUSTRUP, 2000. *The C++ Programming Language*. Addison Wesley Professional. ISBN 0201700735.
- [TimeSys, 2006] TIMESYS CORPORATION, 2006: *RTSJ Login*. <http://www.timesys.com/java/> (September 23, 2006)
- [WOLF, 2001] Wayne WOLF, 2001. *Computers As Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, San Francisco. ISBN 1-55860-541-X.
- [WOLF, 2005] Wayne WOLF (ed), 2005. *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*. ACM. ISBN 1-59593-091-4.
- [ZERZELIDIS, 2005] Alexandros ZERZELIDIS, Andy J. WELLINGS, 2005: *Requirements for a real-time .NET framework*. in *SIGPLAN Notices* vol. 40 no. 2, p 41-50. <http://doi.acm.org/10.1145/1052659.1052666>
- [ZURAWSKI, 2005] Richard ZURAWSKI (ed), 2005. *The Industrial Information Technology Handbook*. CRC Press. ISBN 0-8493-1985-4.
- [CORSARO, 2002a] Angelo CORSARO, Douglas C. SCHMIDT, 2002. *Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems*. in *IEEE Real Time Technology and Applications Symposium [RTAS, 2002]*, p 90-100. ISBN 0-7695-1739-0.

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem einzelnen Fall durch die Angabe der Quelle als Entlehnung kenntlich gemacht.

Berlin, September 29, 2006