



**Università degli Studi di Pisa**

---

FACOLTÀ DI INGEGNERIA

Corso di Laurea Specialistica in Ingegneria Informatica

TESI DI LAUREA SPECIALISTICA

# **Design and development of a nesC to C language translator for the Erika real-time kernel**

Candidato:

**Luca Santocono**

Relatori:

**Prof. Paolo Ancilotti**

**Prof. Giuseppe Lipari**

**Prof. Giuseppe Anastasi**



# Abstract

The nesC programming language is an extension to the C language designed and developed specifically for TinyOS, an operating system for wireless sensor networks. For that reason it tries to reflect TinyOS's event driven and component based architectural model by offering constructs and statements that can be directly mapped to TinyOS components or primitives. Consequently, the existing nesC compiler, which is actually more a translator than a compiler, since the transformation to a low level language is done by the underlying C compiler, is tailored for a use with TinyOS.

This work tries to decouple nesC from TinyOS with the design and realization of a translator which has a double purpose. On the one hand it translates nesC to C and on the other hand it substitutes TinyOS constructs and primitives with Erika ones. Therefore, apart from being a language translator, the resulting software of this work can be seen as an operating system translator as well.

Erika is a wireless sensor network operating system that adds cutting edge real time scheduling algorithms to an OSEK/VDX compliant kernel. The OSEK/VDX standard guarantees to an operating system hardware, application and network independence. To facilitate the porting of new and existing applications to the Erika operating system the present language and operating system translator has been realized from scratch with the help of the JavaCC parser generator which generates appropriate Java classes from a nesC language grammar specification.



*For my father,  
his infinite patience,  
his answers that  
he always had  
for any question.*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Definition . . . . .	1
1.2	State of the art . . . . .	2
1.2.1	nesC TinyOS . . . . .	2
1.2.2	nesC Eclipse plug-ins . . . . .	3
1.2.3	MeshC . . . . .	3
1.2.4	nCUnit . . . . .	3
1.3	Innovations introduced by this thesis . . . . .	3
1.4	Terminology . . . . .	4
1.5	Thesis structure . . . . .	5
<b>2</b>	<b>TinyOS</b>	<b>7</b>
2.1	Design . . . . .	7
2.2	Hardware Abstraction . . . . .	8
2.3	Scheduler . . . . .	9
2.4	Memory model . . . . .	10
2.5	Components . . . . .	11
2.5.1	Commands . . . . .	12
2.5.2	Events . . . . .	12
2.5.3	Tasks . . . . .	13
<b>3</b>	<b>Erika</b>	<b>15</b>
3.1	OSEK/VDX compliant design . . . . .	16
3.2	Architecture . . . . .	18
3.3	Scheduler . . . . .	19
3.4	Memory model . . . . .	19
3.5	RT-Druid . . . . .	20
3.6	nesC to OIL translations . . . . .	22
3.6.1	task . . . . .	22
3.6.2	Timer.fired . . . . .	22

<b>4</b>	<b>The nesC language and its C translation</b>	<b>25</b>
4.1	Component specification . . . . .	25
4.2	Modules and Interfaces . . . . .	28
4.2.1	Split phase operations . . . . .	28
4.2.2	nesC Keywords and their translation . . . . .	29
4.3	Configurations and Wiring . . . . .	34
4.3.1	Implicit Wiring . . . . .	37
4.3.2	Parameterized Wiring . . . . .	38
<b>5</b>	<b>General compiler design</b>	<b>41</b>
5.1	Front end . . . . .	41
5.2	Back end . . . . .	43
5.3	JavaCC - a parser generator . . . . .	44
5.4	JJtree - a parse tree generator . . . . .	44
<b>6</b>	<b>Architecture</b>	<b>47</b>
6.1	Introduction . . . . .	47
6.2	Software Requirements specification . . . . .	47
6.2.1	Users of the software . . . . .	47
6.2.2	Use cases . . . . .	48
6.2.3	Requirements definition . . . . .	49
6.2.4	System requirements specification . . . . .	49
6.3	System Design . . . . .	51
6.3.1	Activities . . . . .	51
6.3.2	Realization of a nesC to C translator . . . . .	52
6.3.3	Realization of a nesC library for Erika . . . . .	59
6.3.4	Version control . . . . .	60
6.3.5	Testing software . . . . .	60
6.3.6	Selecting a cross compiler for target platforms . . . . .	61
6.4	Software design . . . . .	61
6.4.1	nesC to C translator design . . . . .	61
6.4.2	Realization of a nesC library for Erika . . . . .	65
6.5	Implementation details . . . . .	66
6.5.1	The front end . . . . .	67
6.5.2	The parser . . . . .	67
6.5.3	The C and OIL code generators . . . . .	67
6.5.4	The symbol table . . . . .	67
6.5.5	XML parsers . . . . .	68
6.5.6	The nesC entity Objects . . . . .	68
6.6	Preparing of the testing software . . . . .	69



<b>7</b>	<b>Installation and Using</b>	<b>75</b>
7.1	Software Requirements . . . . .	75
7.2	Installation . . . . .	76
7.2.1	Installation of nesC-Erika . . . . .	76
7.3	Using - Sample compilation on an AVR target . . . . .	77
<b>8</b>	<b>Experiments</b>	<b>85</b>
8.1	Target Hardware . . . . .	85
8.1.1	STK500 development board . . . . .	85
8.1.2	STK501 development board . . . . .	87
8.1.3	ATmega128 AVR microcontroller . . . . .	88
8.2	In-System Programmer . . . . .	89
8.3	Cross compiler . . . . .	90
8.4	nesC-TinyOS vs nesC-Erika . . . . .	90
<b>9</b>	<b>Future work</b>	<b>97</b>
9.1	Erika nesC library . . . . .	97
9.2	Going away from nesC-TinyOS . . . . .	97
9.3	TinyOS 2 support . . . . .	98



# List of Figures

2.1	Blink component graph . . . . .	8
2.2	Simplified architecture of a TinyOS application . . . . .	9
2.3	A scheduling example . . . . .	10
2.4	A frame of a TinyOS component . . . . .	11
2.5	Division of the components based on its functionality . . . . .	12
3.1	Example development process for applications . . . . .	17
3.2	Architecture of the Erika OS . . . . .	19
3.3	HAL monostack . . . . .	20
3.4	HAL multistack . . . . .	20
3.5	RT-Druid file generation from an OIL file . . . . .	21
5.1	A typical compiler architecture . . . . .	42
5.2	File generation process of JavaCC . . . . .	45
5.3	Class hierarchy representing the parse tree . . . . .	46
6.1	Gantt chart of the different activities . . . . .	52
6.2	Translation process . . . . .	53
6.3	Front end system design . . . . .	55
6.4	Parser system design . . . . .	56
6.5	Error checker system design . . . . .	58
6.6	C code generator system design . . . . .	59
6.7	OIL code generator system design . . . . .	60
6.8	Compilation process . . . . .	62
6.9	Front end software design . . . . .	63
6.10	C code generator software design . . . . .	65
6.11	OIL code generator software design . . . . .	66
6.12	The visitors class hierarchy . . . . .	68
6.13	Symbol table class diagram . . . . .	69
6.14	The SAX handler hierarchy . . . . .	69
6.15	The NESCentity class hierarchy . . . . .	70

7.1	Cygwin shell . . . . .	77
7.2	Entering the directory of the downloaded package . . . . .	78
7.3	Extracting the archive . . . . .	79
7.4	The nesC-Erika installation directory tree . . . . .	80
7.5	The application Makefile . . . . .	81
7.6	Entering the nesC-Erika installation directory . . . . .	81
7.7	The ncc Perl script . . . . .	82
7.8	Compiling the sample application . . . . .	82
7.9	Message after successful compilation . . . . .	83
7.10	Files generated by the translator . . . . .	83
7.11	The <i>avr.hex</i> file located in the <i>nesc_erika/Demo/BlinkTask/Debug</i> directory. . . . .	84
7.12	Using the script <i>sample_uisp_script.sh</i> . . . . .	84
8.1	The STK500 board . . . . .	86
8.2	The STK501 board on top of the STK500 . . . . .	88

# List of Tables

8.1	Files generated by nesC-Erika for Blink . . . . .	91
8.2	Files generated by nesC-TinyOS for Blink . . . . .	91
8.3	nesC-TinyOS vs. nesC-Erika intermediate files for Blink . . .	92
8.4	nesC-TinyOS vs. nesC-Erika executables for Blink . . . . .	92
8.5	Files generated by nesC-Erika for BlinkTask . . . . .	92
8.6	Files generated by nesC-TinyOS for BlinkTask . . . . .	93
8.7	nesC-TinyOS vs. nesC-Erika intermediate files for BlinkTask .	93
8.8	nesC-TinyOS vs. nesC-Erika executables for BlinkTask . . . .	94
8.9	Files generated by nesC-Erika for CntToLeds . . . . .	94
8.10	Files generated by nesC-TinyOS for CntToLeds . . . . .	94
8.11	nesC-TinyOS vs. nesC-Erika intermediate files for CntToLeds	94
8.12	nesC-TinyOS vs. nesC-Erika executables for CntToLeds . . . .	95
8.13	Files generated by nesC-Erika for GlowLeds . . . . .	95
8.14	Files generated by nesC-TinyOS for GlowLeds . . . . .	95
8.15	nesC-TinyOS vs. nesC-Erika intermediate files for GlowLeds .	95
8.16	nesC-TinyOS vs. nesC-Erika executables for GlowLeds . . . .	96



# Chapter 1

## Introduction

The aim of the present thesis is to realize an automatized tool able to translate applications written in the nesC language for the TinyOS operating system to applications written in C for the Erika operating system. In the first instance in this chapter the problem is introduced. After that, the context in which the software can be used is described and then the innovations brought by this thesis are analyzed. Finally, the structure of this document is presented.

### 1.1 Problem Definition

TinyOS is one of the most used operating systems for wireless sensor networks. Hence, there are a lot of applications available written for it using the nesC programming language. Having a tool that automatically translates them to the C language ready to be used with the Erika operating system, is a great opportunity for both the operating system by increasing the number of possible target users and for the programmer of the application who has an easy way to transform it to a real time application. To achieve this goal a so called translator needs to be realized. It is mostly quite the same tool as a compiler, the main difference is in the output language. A translator, also called language converter, transforms a high level programming language to another high level programming language, whereas a compiler converts a language to a low level programming language, such as assembly or machine language, ready to be executed on a target microprocessor or microcontroller. In this case the translation happens between the nesC and C programming languages, both of them are high level languages not ready to be executed on a target microprocessor.

The tool presented in this thesis is a little bit different, because it is not only a language converter, but an operating system translator as well.

Accordingly, it has to integrate in one software tool the two different aspects of translating a programming language and an operating system to another one. In this context, translating the operating system to another one means to realize an automatic substitution of the system calls of the first operating system to the system calls of the second one. In this case, the two operating systems in question are TinyOS and Erika.

Two main difficulties arise in doing the translation of one operating system to another one. The first problem is caused by the lack of equivalent system calls between the two operating systems. Another difficulty is that the set of target boards and target microcontrollers are different in the two operating systems. Consequently, there is sometimes also the need to translate some target platform specific statements, like for example some *asm* statements<sup>1</sup>. Besides trying to identify those problems, this document attempts to give suggestions on how to solve them and the realized software represents a prototype of how the automatized translation can be done.

## 1.2 State of the art

At the time of writing this document there are a few projects, besides the classic nesC compiler [NES] developed for TinyOS that are trying to realize a compiler for the nesC programming language. In this section a brief description of this projects is given.

### 1.2.1 nesC TinyOS

This is the first nesC compiler, it was developed with the definition of the nesC language. New versions with bugfixes and even new constructs are released on a regular basis. The current newest version is nesC v1.3, released on August, 6th 2008. The previous major version nesC v1.2 introduced a lot of new constructs compared with the nesC v1.1 series. Particularly, the whole TinyOS operating system was completely rewritten with nesC v1.2 leading to the release of TinyOS 2.

Because it is maintained by the inventors of the nesC language, this is for sure the compiler that should be the reference for all other attempts of writing a new nesC compiler from scratch.

---

<sup>1</sup>Assembly language statements integrated in a C program



### 1.2.2 nesC Eclipse plug-ins

There are quite a few nesC plug-ins for Eclipse, like for example YETI 2 [YET], TinyDT [TDT] and TinyosIDE [TID], some of them working for nesC v1.1 and others for nesC v1.2.

Actually, these are neither real translators nor real compilers, because they are actually only parsing the source code for syntax highlighting and other similar purposes leaving the code generation to the nesC compiler.

Another interesting project is Cadena [CAD] that implements a complete high level modelling environment integrated into Eclipse that permits to develop applications by drawing block diagrams. It permits to generate nesC code from those diagrams and to import nesC code to produce such diagrams. Therefore there is a nesC parser integrated, but it parses nesC code and produces block diagrams as output and not C code.

So it can be seen that the aim of those project is different that the purpose of the project described in this document.

### 1.2.3 MeshC

MeshC [MES] is an interrupted project that was aiming not only to produce a nesC compiler that is independent from the operating system, but that is out-and-out an extension to the nesC language defining in fact a new language, which is backwards compatible with nesC v1.1.

### 1.2.4 nCUnit

nCUnit [NCU] is a unit testing framework for nesC. nCUnit uses a pre-compiler that inserts calls to the test case functions, which are tagged with the "@test()" attribute. By modifying a constant in the compiled code, one of the test functions is selected for each simulation run. In addition, by running its own processor between the nesC and the avr-gcc compilers, it modifies the declaration of functions that are monitored using the "assertCalls" assertion. The aim of this project is again different, since it consists in unit tests for code that can be used to execute other code bodies outside the application in question.

## 1.3 Innovations introduced by this thesis

Compared with the projects shown in section 1.2, the present project has quite a different purpose. On the one side it tries to clone the behaviour

of the original nesC-TinyOS compiler introduced in section 1.2.1 by implementing a nesC parser and C code generator, on the other side it extends this functionality by adding also an operating system translator to it. This means that it is not enough to produce C code as output, but also to substitute some TinyOS specific functions with Erika specific ones. Using Erika specific constructs means that, besides generating C code it is also needed to produce OIL [OSE] code. Consequently, besides the normal use, this project can serve also as a starting point for the development of new nesC compilers, maybe for other operating systems, because the source code is quite comprehensible compared with the nesC-TinyOS source code, which is written extending the gcc compiler collection [GNU].

## 1.4 Terminology

For a better understanding of this document some explanations about the used terminology are necessary. The following list explains the meaning of some terms that can lead to confusion.

**Compiler, translator** Compiler and translator are almost the same thing and sometimes both terms are used for the same entity in this document. Even in the literature the difference between them is not that clear. The most accepted definition to distinguish them is that a compiler transforms a high level language to assembly or machine language ready to be executed on a target CPU, whereas a translator converts a high level language to another high level language. Nevertheless, there can be found some Assembly translators that translates an assembly language in another one and with the above definition they would neither be counted as translators nor as compilers, since they are translating a low level language to another low level language ready to be executed on a target CPU <sup>2</sup>.

**nesC-Erika installation directory** This is the absolute path to the nesC-Erika software once installed on a system. Section 7.2.1 explains in detail how to install the software.

**nesC-TinyOS, original nesC compiler** This is the classic nesC compiler [NES] developed when the nesC language was introduced.

---

<sup>2</sup>The definition that I personally prefer is the following:

A compiler compiles a higher level language to a lower level one.

A translator translates a language to another one of the same level.

**nesC-Erika** This is the translator developed during this project.

**main C file** This refers to the C file generated by the nesC-Erika translator that contains the main function and almost all the other function definitions.

## 1.5 Thesis structure

The document introduces firstly the two operating systems on which the work is based on, by trying to emphasize the differences between their organization. Consequently, chapter 2 begins with a description of the TinyOS operating system by explaining the design philosophy, the available hardware abstraction, the scheduler and the memory model. Similarly, chapter 3 describes the same aspects for the Erika operating system with a final section (3.6), where some translations from nesC to OIL are shown. Chapter 4 tries to outline the most important concepts in the nesC language and shows in addition their translation to the C language. After that, chapter 5 summarizes the main aspects of compiler design theory, introduces JavaCC, the chosen parser generator and JJtree the chosen parse tree generator. Afterwards, chapter 6 describes in detail the various development phases of the project by analyzing in the first instance the software requirements specification, then the system level design, the software design, some implementation details and finally the testing methodology. Chapter 7 describes a method for the installation of the nesC-Erika software and shows an example usage on an AVR target microcontroller. Chapter 8 compares the realized nesC-Erika compiler with the original nesC-TinyOS compiler by reporting the different memory sizes and line numbers of the generated files. Last but not least, chapter 9 gives some suggestion and ideas on how to continue the work on the software.



# Chapter 2

## TinyOS

TinyOS is an event driven component based open source operating system for wireless sensor networks. Since v1.x it is written using the nesC language, which was specifically designed for it. The latest version of TinyOS is v2.x. Many components has been written in a new way and are renamed passing from 1.x to 2.x. In particular the 2.x version uses some constructs that are available only in the newer nesC v1.2.x language, while the 1.x version of TinyOS uses a pure nesC v1.1.x syntax. Since in literature normally a description of TinyOS in conjunction with an explanation of the nesC language is found, this chapter focuses more on the description of TinyOS as an operating system itself trying to decouple it from the explanation of the nesC language, which is explained in chapter 4.

### 2.1 Design

A TinyOS application can be represented by a graph of components and a scheduler. The scheduling model consists of two levels: tasks and events. A component consists of a frame<sup>1</sup> for storage purposes, tasks for making the concurrency possible and events<sup>2</sup> that can be divided in commands and handlers. Figure 2.1 shows the components graph of the Blink demo application as example.

---

<sup>1</sup>see section 2.4 for more details about the meaning of frame

<sup>2</sup>not to be confused with the nesC keyword.

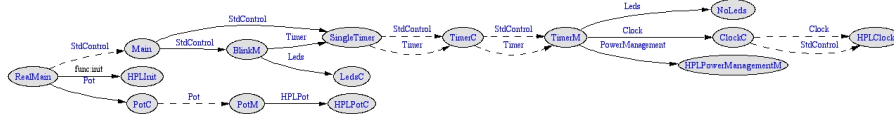


Figure 2.1: Blink component graph

## 2.2 Hardware Abstraction

Hardware abstractions in TinyOS 2.0 generally follow a three level abstraction hierarchy, called the HAA (Hardware Abstraction Architecture).

At the bottom of the HAA the HPL (Hardware Presentation Layer) can be found. The HPL is a thin software layer on top of the raw hardware, presenting hardware such as IO pins or registers as nesC interfaces. The HPL generally has no state besides the hardware itself. This means that it has no variables. HPL components usually have the prefix `Hpl`, followed by the name of the chip. For example, if a chip is called `CC1000`, the HPL components of the chip begin with `HplCC1000`.

The middle of the HAA is the HAL (Hardware Abstraction Layer). The HAL builds on top of the HPL and provides higher-level abstractions that are easier to use than the HPL, but still provide the full functionality of the underlying hardware. The HAL components usually have a prefix of the chip name. For example, the HAL components of the `CC1000` begin with `CC1000`.

The top of the HAA is the HIL (Hardware Independent Layer). The HIL builds on top of the HAL and provides abstractions that are hardware independent. This generalization means that the HIL usually does not provide all of the functionality that the HAL can. HIL components have no naming prefix, as they represent abstractions that applications can use and safely compile on multiple platforms. For example, the HIL component of the `CC1000` on the `mica2` is `ActiveMessageC`, representing a full active message communication layer. Some components may not have an implementation on the HIL level and therefore their implementation on a lower level (HAL or HPL) needs to be used.

The resulting architecture of an application written for TinyOS can be seen in figure 2.2. In the picture the three hardware abstraction layers are summarized in one stripe to simplify the representation.

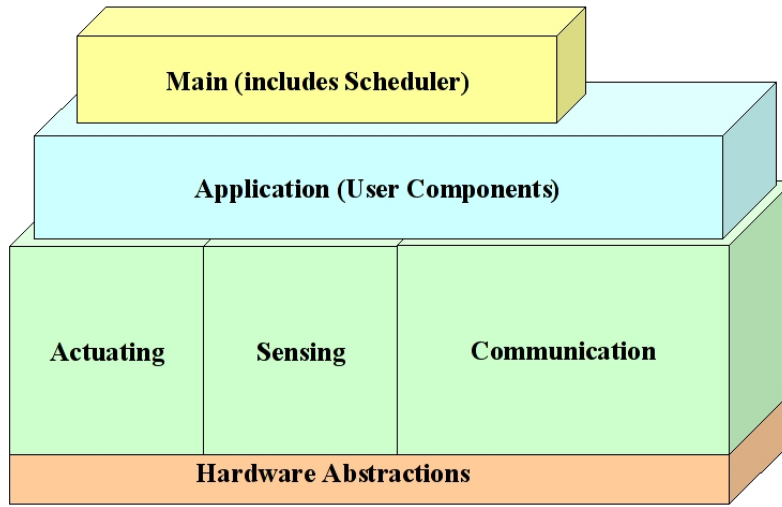


Figure 2.2: Simplified architecture of a TinyOS application

## 2.3 Scheduler

Both the TinyOS 1.x and the TinyOS 2.x scheduler have a non-preemptive FIFO policy. However, tasks in 2.x operate slightly differently than in 1.x. In TinyOS 1.x, there is a shared task queue for all tasks, and a component can post a task multiple times. If the task queue is full, the post operation fails. Experience with networking stacks showed this to be problematic, as the task might signal completion of a split-phase operation (see 4.2.1): if the post fails, the component above might block forever, waiting for the completion event.

In TinyOS 2.x, every task has its own reserved slot in the task queue, and a task can only be posted once. A post fails if and only if the task has already been posted. If a component needs to post a task multiple times, it can set an internal state variable so that when the task executes, it reposts itself.

This slight change in semantics greatly simplifies a lot of component code. Rather than test to see if a task is posted already before posting it, a component can just post the task. Components do not have to try to recover from failed posts and retry.

Figure 2.3 shows an example, where hardware interrupts are raising events. Those events can preempt a task, if one is running and they can call some commands. Commands might post tasks that means that they are putting

tasks in the ready queue.

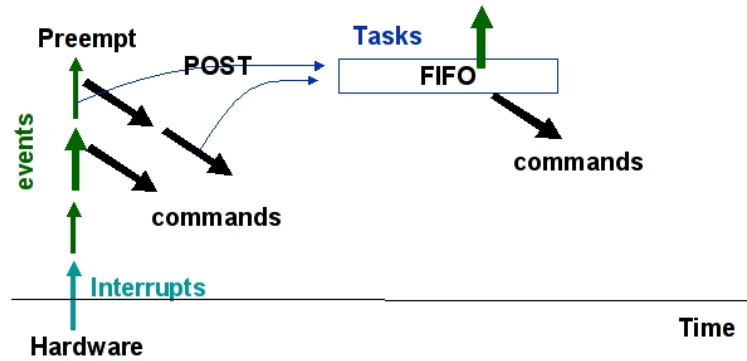


Figure 2.3: A scheduling example

Applications can also replace the scheduler, if they wish. This allows programmers to try new scheduling policies, such as priority- or deadline-based. It is important to maintain non-preemptiveness, however, or the scheduler will break all nesC's static concurrency analysis.

Finally it is to note that the scheduler puts the processor, but not the peripherals to sleep, if the active task queue is empty.

## 2.4 Memory model

In TinyOS there is no difference between kernel and user space. In addition it has a static memory allocation. This means that no heap is available and therefore C's malloc cannot be used. Moreover, TinyOS has no virtual memory so a single linear physical address space is available for the allocation of the memory assigned to each component. Every component has an own frame of size 4K in a shared global stack. The frame is allocated to the application at compile time. Global variables are available on a per-frame basis. A frame has three parts: a stack that contains local variables declared within a method, a global part for storing the global variables of a component not declared in any method and a free part, if the frame is not full. Figure 2.4 gives an idea of this structure.



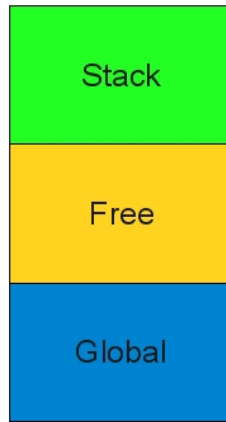


Figure 2.4: A frame of a TinyOS component

## 2.5 Components

The whole TinyOS operating system is divided into components. Figure 2.5 shows an example application that is divided in components that have different functionalities. The arrows that are pointing to the top direction are presenting events schematically, while the arrows that are pointing to the bottom are representing commands. The various components are grouped into categories responsible for the implementation of different functionalities.

Components use and provide interfaces, commands and events<sup>3</sup>. Components can be viewed as finished state machines, in that command and event handlers transition a component from one state to another. This leads to low overhead and non-blocking state transitions.

There are different types of components that can be classified by the abstraction layer to which they belong or by the functionality that they implement.

- Hardware abstraction components for controlling i.e.:
  - Leds
  - Clock
  - UART (Universal Asynchronous Receiver Transmitter)
- Synthetic hardware components able to:

---

<sup>3</sup>see chapter 4 for the corresponding nesC keywords

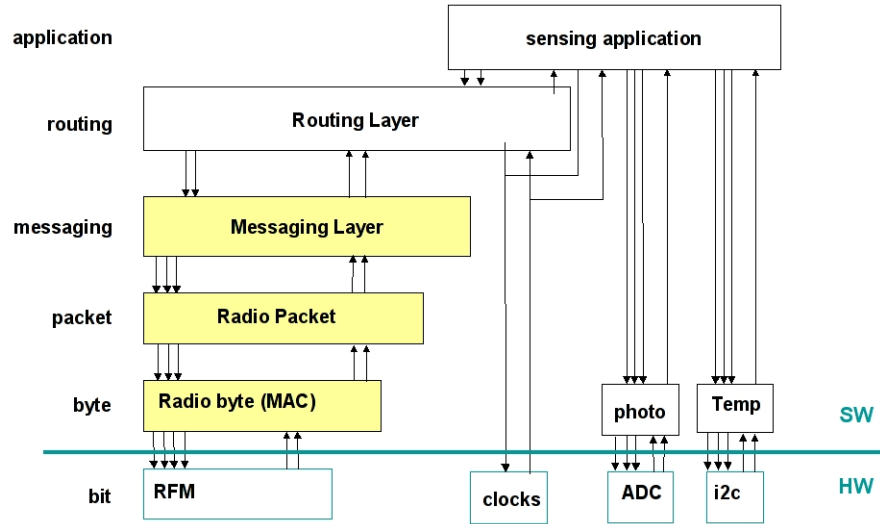


Figure 2.5: Division of the components based on its functionality

- simulate hardware behavior
- enhance the state machine behaviour
- High-level software components, that:
  - perform control, routing and all data transformations

### 2.5.1 Commands

Commands consist in non-blocking requests made generally to lower level components. They provide the caller with feedback by returning status, implementing that way a kind of callback. In addition commands may post tasks or call other commands, but they are not permitted to signal events.

### 2.5.2 Events

Events signal upward to notify that an action has occurred. It is a non-blocking signal. Lowest-level events are triggered by hardware interrupts,

timer events, or counter events. Events can signal other events, post tasks and even call commands.

### **2.5.3 Tasks**

Tasks are responsible for doing the primary computation work. They are atomic and cannot be preempted by other tasks. Only interrupt handlers, like certain events, can interrupt tasks. A single stack that is assigned to the current running task is available to store the needed informations. Tasks are able to call commands, signal events, and schedule other tasks within a component.



# Chapter 3

## Erika

Erika Enterprise is a multi-processor, real-time operating system kernel, which is available for several platforms ranging from Atmel's AVR5 to the dsPIC (R) DSC microcontrollers family.

Erika Enterprise offers the availability of real-time schedulers and resource managers allowing a fully multithreaded environment, while guaranteeing predictable real-time performance and retaining the programming model of conventional single processor architectures.

The advanced features provided by Erika Enterprise are:

- Support for four conformance classes to match different application requirements;
- Support for preemptive and non-preemptive multitasking;
- Support for fixed priority scheduling;
- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage;
- Support for shared resources;
- Support for periodic activations using alarms;
- Support for centralized Error Handling;
- Support for hook functions before and after each context switch;

The Erika Enterprise kernel has been developed with the idea of providing the minimal set of primitives which can be used to implement a multithreading environment. The Erika Enterprise APIs are implemented as a reduced

set of OSEK/VDX APIs, providing support for thread activation, mutual exclusion, alarms, and counting semaphores.

The OSEK/VDX consortium provides the OIL language (OSEK Implementation Language) as a standard configuration language, which is used for the static definition of the RTOS objects which are instantiated and used by the application. Erika Enterprise fully supports the OIL language for the configuration of real-time applications.

Erika Enterprise is natively supported by RT-Druid, a tool suite for the automatic configuration and deployment of embedded applications which enables to easily exploit multi processor architectures and achieve the desired performance without modifying the application source code.

### 3.1 OSEK/VDX compliant design

OSEK/VDX is a joint project of the automotive industry that aims to the definition of an industry standard for an open ended architecture for distributed control units in vehicles. The objective of the standard is to describe an environment which supports efficient utilization of resources for automotive control unit application software. This standard can be viewed as a set of API for real-time operating system (OSEK) integrated on a network management system (VDX) that together describes the characteristics of a distributed environment that can be used for developing automotive applications.

The typical applications that have to be implemented have tight real-time constraints and a high criticality, like for example, a power-train application. Moreover, these applications have to be made in a huge number of units, therefore there is a need to reduce the memory footprint to a minimum, enhancing as possible the OS performance. Figure 3.1 shows a development cycle of an application. This cycle can change slightly on the different implementations of the operating system, in Erika for example there is no OSEK builder, so the OIL file is edited by hands. In Erika the system generator's work is done by RT-Druid.

Here are the main characteristics of an OSEK/VDX compliant operating system:

- Scalability

The operating system is intended for use on a wide range of control units. To support a wide range of systems the standard defines four conformance classes that tightly specifies the main features of an OS. Note that memory protection is not supported at all.

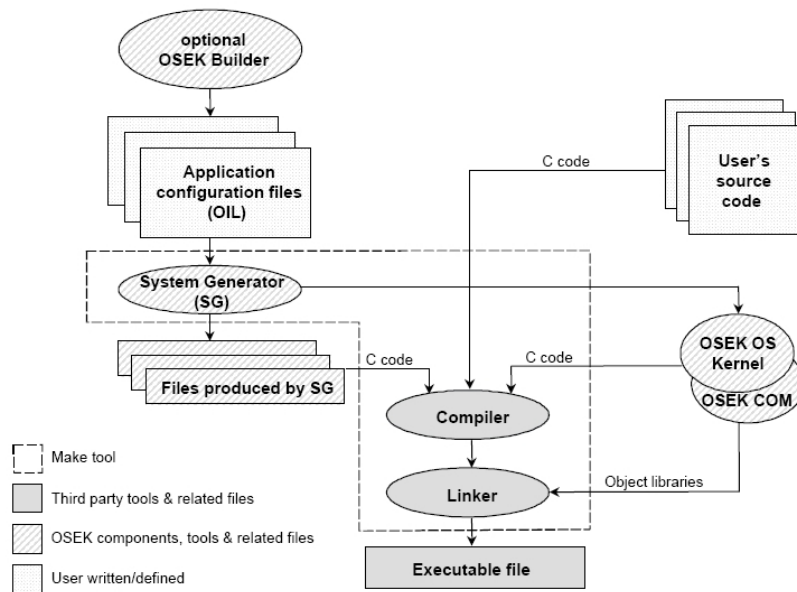


Figure 3.1: Example development process for applications

- Portability of software

The standard specifies an ISO/ANSI-C interface between the application and the operating system that is identical in all the implementations of the OS. The aim of this interface is to give the ability to transfer an application software from one electronic control unit (ECU) to another ECU without bigger changes inside the application.

- Configurability

Another prerequisite needed to adapt the OS to a wide range of hardware is a high degree of modularity and configurability. This configurability is reflected by the toolchain proposed by the OSEK standard, where some configuration tools help the designer in tuning the system services and the system footprint. Moreover, a language called OIL (OSEK Implementation Language) is proposed to help the definition of a standardized configuration information.

- Statically allocated OS.

All the OS objects and features are statically allocated. This fact allow to simplify all the OS: the number of application tasks, resources and

services requested are defined at compile time. Note that this approach ease the implementation of an OS capable of running on ROM, and furthermore it is completely different from a dynamic approach followed in other OS standards like for example POSIX.

- Support for time triggered architectures

The OSEK Standard provides the specification of OSEKTime OS, a time triggered OS that can be fully integrated in the OSEK/VDX framework.

After this brief introduction regarding the OSEK/VDX standard the following sections explain the organization of the Erika operating system.

## 3.2 Architecture

Erika has a three level architecture as depicted in figure 3.2 At the top the application layer can be found consisting in the code written by the programmer.

In the middle the kernel layer is located, which consists in a set of modules that are responsible for the task and real time management. The offered functions can be divided in:

- Queue handling
- Scheduling
- Application programming interface (API)

At the bottom of the hierarchy, the HAL (hardware abstraction layer) is present, which is further divided in the following components:

- MCU layer (microcontroller unit layer)
- CPU layer (central processing unit layer)
- Board layer



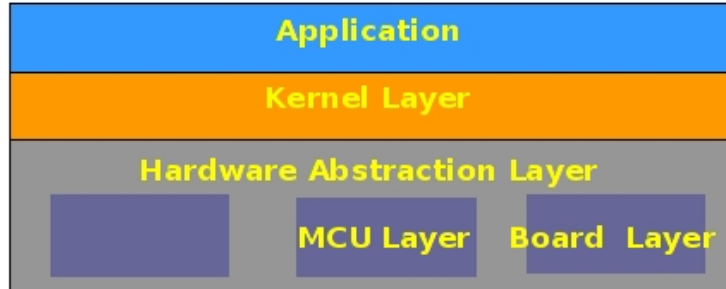


Figure 3.2: Architecture of the Erika OS

### 3.3 Scheduler

Two types of schedulers are included in the kernel: fixed priority (FP) and earliest deadline first(EDF).

Fixed priority scheduling, allows users to set fixed priorities on a each task of the application. In the default configuration, the highest priority task always gets the CPU as soon as it is runnable, even if other system tasks are in execution in that moment. In this case the task that is being executed is preempted and the task with the highest priority is executed.

Earliest deadline first scheduling in contrast is a dynamic scheduling algorithm. It places tasks in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.), the queue will be searched for the process closest to its deadline. This process will then be the next one to be executed.

### 3.4 Memory model

Two different memory model paradigms are available in Erika for the organization of an application.

With the first one, called HAL monostack and represented in figure 3.3, tasks and services routines are sharing the same stack.

With the second one, called HAL multistack and shown in figure 3.4, each task and each service routine has its on reserved stack.

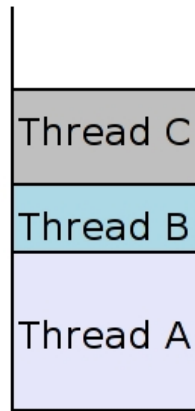


Figure 3.3: HAL monostack

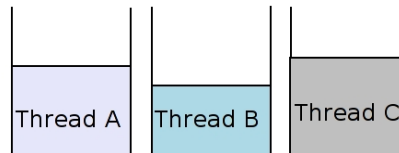


Figure 3.4: HAL multistack

### 3.5 RT-Druid

RT-Druid is an open and extensible environment, based on XML and open standards (Java) allowing generation of portable OSEK C code from OIL definitions to create applications that run in real-time in a variety of environments, including ARM7, PPC, ST10 and the Altera Nios II Softcore.

Generated code can run on any OSEK-compliant system, but the RT-Druid framework is optimized for running in conjunction with the Erika Enterprise kernel. Because of its generic framework, RT-Druid gives an extensible modeling and analysis platform for modeling any hardware and software, providing compatibility with most of the model-based methodologies for functional design.

Giving an OIL configuration file as input, the RT-Druid tool creates a directory which contains the generated files presented schematically in figure 3.5.

The first file that is created is the makefile. The makefile is used to

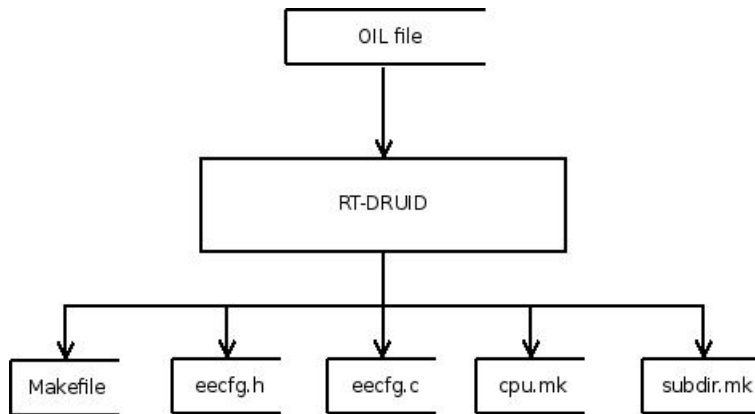


Figure 3.5: RT-Druid file generation from an OIL file

compile the application source code. The makefile structure may depend on the final target architecture.

The other generated files are:

- `eecfg.h`

This file contains the declarations of all the RTOS symbols (tasks, resources, alarms, events, and so on) that are visible from the given CPU. The objects visible from a CPU are the objects allocated on it, plus the objects on other CPUs that may be referred by the code running on the CPU itself.

- `eecfg.c`

This file contains the configuration data structures of the Erika Enterprise kernel, providing information on the OIL file local objects options.

- `cpu.mk`

This file contains the rules used to compile the source code allocated to the CPU.

- `subdir.mk`

This file contains the list of the files that must be compiled and linked in order to generate the executable to be run on the CPU. The files depend on the partitioning configuration defined in the OIL file.

## 3.6 nesC to OIL translations

Some nesC constructs need a translation in OIL rather than in C to permit a better integration with Erika and its philosophy. In particular at the time of writing this document two nesC constructs are translated into OIL constructs: the *task* keyword and the *Timer.fired* event that is signaled by a timer interrupt. Its translation is described below.

### 3.6.1 task

A nesC declaration like

```
task void task_A () {
    //some instructions
}
```

leads to the generation of entries in three files. The generated entry in the main C file can be seen in section 4.2.2.

The

```
TASK task_A ();
```

entry is generated in the *handler.c* file.

Finally, the entry

```
TASK A{
    PRIORITY = 1;
    ACTIVATION = 4;
    STACK = SHARED;
    SCHEDULE = FULL;
};
```

is generated in the OIL configuration file of the application. The various options are for now statically inserted like above. In the future it may be possible to change the values, but for doing so some extensions to the nesC language are needed.

### 3.6.2 Timer.fired

Each use of this TinyOS system call results in three generated entries one in the main C file, one in the *handler.c* file and the final one in the OIL configuration file.

Therefore a piece of code like

```

event result_t Timer.fired()
{
    //some instructions
}

```

leads to the following translation in the main C file.

```

result_t Timer__fired()
{
    //some instructions
}

```

Furthermore, it leads to the following translation in the OIL configuration file.

```

HANDLER = HANDLER_T1_OVERFLOW {
    FUNCTION = "irq_f__type1";
    TYPE = 2;
};

```

The *irq\_f\_\_type1* routine is executed, when an overflow occurs on timer 1.

Finally, the *irq\_f\_\_type1* routine needs to be defined and therefore the following entry in the *handler.c* file is generated.

```

void irq_f__type1(void) {
    BlinkTaskM__Timer__fired();
}

```

*BlinkTaskM\_\_Timer\_\_fired()* is a function that is defined in the TinyOS application that is going to be translated. Normally, it calls another function that is the actual interrupt handler.



# Chapter 4

## The nesC language and its C translation

In this chapter an overview of the nesC language is given. In particular it focuses on how to translate the various nesC constructs into the C language. Since there is no code generation support for nesC v1.2.x constructs, only nesC v1.1.x constructs are described.

For a more theoretical and formal nesC language reference [GLCB03] and [GLCB05] can be seen. For a practical introduction on how to code in nesC [Lev06] is very useful.

Firstly, it is to note that the nesC language can be divided into two main entities: components and interfaces. A component is either a module or a configuration, whereas an interface is some kind of bidirectional entity that makes it possible to connect a component with another one.

### 4.1 Component specification

A component's specification is the set of interfaces that it provides and uses. Each provided or used interface has a name and an interface type. The interface type matches the name of the interface used in its definition. Component specifications can also contain *bare* commands and events. These are not contained in any interface. In addition typedefs and tagged type declarations, like enums, can be included in a component's specification as well.

Three specific nesC keywords are used to specify the component's specification, *provides*, *uses* and *as*. The last of these can be omitted. The typical syntax for writing a component specification is as follows.

**uses** *interface-type* **as**<sub>opt</sub> *instance-name*

or

**provides** *interface-type* **as**<sub>opt</sub> *instance-name*<sub>opt</sub>,

where *interface-type* is the name used in the interface definition and *instance-name* is an arbitrary identifier.

**provides** The *provides* keyword is used to declare the provided interfaces or bare commands. If it declares an interface, all commands that are present in that interface need to be defined, otherwise a compile-time error is thrown. Events contained in the provided interface can be signaled.

**uses** The *uses* keyword is used to declare the used interfaces or bare events. If it declares an interface, all events that are present in that interface need to be defined, otherwise a compile-time error is thrown. Commands contained in the used interface can be called.

**as** The *as* keyword is used to rename the interfaces used in the component, allowing for example, to declare the same interface as used and as provided in the same component. If the *as* keyword is omitted, the *interface-type* and the *instance-name* are the same, and these are the same as the name used in the interface definition.

For instance, the following is a valid component specification.

```
//File A.nc

module A {
    uses interface X;
}

implementation {
    ...
}

//End of file A.nc

//File B.nc
```



```

module B {
    provides interface X as Y;
}

```

```

implementation {
    ...
}

```

*//End of File B.nc*

*//File X.nc*

```

interface X {
    command int c();
    event int e();
    ...
}

```

*//End of file X.nc*

Since the module A is *using* the interface X, it has to implement the event  $e()$ , otherwise a compile time error occurs. Same thing for B regarding the command  $c()$  contained in interface X.

*//File A.nc*

```

    ...
implementation {
    event int X.e() {
        ...
        return 1;
    }
}

```

*//End of file A*

*//File B.nc*

```

    ...
implementation {
    command int Y.c(){
        ...
    }
}

```

```

        return 1;
    }
}

//End of file B

```

It is to note that B uses `Y.c()` as identifier for the command, because in the component specification the interface was “renamed” using the *as* keyword.

## 4.2 Modules and Interfaces

Modules embody the program logic of an application in terms of function and task definitions, exactly the same way as contained in a C file. In contrast interfaces contain the declarations of functions that can be defined by modules. Interfaces can either be used or provided by modules (4.1), giving that way some kind of constraints on how modules can define functions, on how they can be wired together (4.3) and allowing so for example to define the concept of split phase operations as described in 4.2.1.

In addition in this section all nesC v1.1.x specific constructs and keywords, that can be found in modules and interfaces are explained. For C ones it is referred to [KR78] and for nesC v1.2 ones [GLCB05] can be seen.

### 4.2.1 Split phase operations

An important concept in the TinyOS operating system and consequently in the nesC language are split phase operations. A split phase operation is a some kind of “third way” of handling operations, besides synchronous or asynchronous operations. It emulates very well the hardware behaviour, which is seldom blocking. Split phase means that completion of a request is a callback. An important characteristic of split-phase interfaces is that they are bidirectional: there is a downcall to start the operation, and an upcall that signifies the operation is complete. For example, to acquire a sensor reading with an analog-to-digital converter (ADC), the software writes to a few configuration registers to start a sample. When the ADC sample completes, the hardware issues an interrupt, and the software reads the value out of a data register. In nesC, downcalls are generally commands, while upcalls are events. An interface specifies both sides of this relationship.

### 4.2.2 nesC Keywords and their translation

In this section the various nesC v1.1 keywords are described with its translation in the C language. There are some nesC keywords that have a meaning only for the nesC compiler and therefore they don't have a C translation.

**async** The *async* storage class specifier must be added on commands and events, which can safely be executed by interrupt handlers, but this does not necessary mean that the defined function must be an interrupt handler.

For example the following is a valid use.

```
interface Leds {
    async command result_t redOn();
    async command result_t redOff();
}
```

In this case the commands redOn() and redOff() can be used in an interrupt handlers, for example a timer interrupt handler.

**post** The *post* keyword followed by a task, puts the task in the ready state and at some point the scheduler can decide to execute it.

For example, the code

```
...
post TaskA;
...
```

is translated in nesC-Erika by,

```
PostTask(TaskA);
```

The *PostTask* function is defined in the *ee\_tinyos.c* file located in the header subdirectory of the Erika installation directory <sup>1</sup>. It is to note that the function, which is shown below, is not an Erika primitive.

```
int PostTask( void (*task) ()) {
    ActivateTask(task);
    return 1;
}
```

---

<sup>1</sup>see sections 1.4 and 7.2.1 for the meaning of nesC-Erika installation directory.

This function was introduced because the nesC-Erika primitive *ActivateTask* at the time of writing this document doesn't return any value, whereas the correspondent nesC-TinyOS primitive returns one of two possible values and can therefore be used for example as a condition in an if statement.

**call** The *call* keyword followed by a command is used to make a command call. Basically, this is exactly the same as a C function call, a command being nothing else than a particular kind of function. Nevertheless this keyword was introduced to allow the signaling of an error, if instead an event is called. An example usage of this keyword can be seen below.

```
...
call aCommand();
...
```

There is no translation in the C language for it, since this keyword has a meaning only for the nesC compiler.

**task** The *task* storage class specifier is added to a function definition to inform the compiler that the following is a task definition and not a simple function.

A typical use of this keyword is shown below.

```
...
task void aTask() {
    //some instructions
}
...
```

The correspondent nesC-Erika translation is listed below.

```
...
TASK aTask() {
    //some instructions
}
...
```

*TASK* is a macro defined in the Erika kernel.

**signal** The *signal* keyword followed by an event is used to make an event call. Basically, this is exactly the same as a C function call, an event being nothing else than a particular kind of function. Nevertheless this keyword was introduced to allow the signaling of an error, if instead a command is signaled.

```
...
signal anEvent();
...
```

There is no translation in the C language for it, since *signal* has a meaning for the nesC compiler only.

**atomic** The *atomic* keyword guarantees that the statement is executed without being interrupted by another computation occurred simultaneously. The simplest way to do this is by disabling the interrupts.

A sample usage of this keyword can be the following.

```
void f() {
    bool available;
    atomic {
        available = !busy;
        busy = TRUE;
    }
    if (available)
        do_something;
    atomic busy = FALSE;
}
```

To note is that it can be used either in conjunction with curly brackets or without. In any case the translation looks like in the following.

```
void f() {
    bool available;

    EE_hal_disableIRQ();
    {
        available = !busy;
        busy = TRUE;
    }
    EE_hal_enableIRQ();

    if (available)
        do_something;

    EE_hal_disableIRQ();
    {
        busy = FALSE;
    }
}
```

```

        EE_hal_enableIRQ ();
    }

```

*EE\_hal\_disableIRQ* and *EE\_hal\_enableIRQ* are two architecture independent Erika primitives used to disabling and enabling the interrupts respectively.

**command** The *command* storage class specifier indicates that the function is part of a component's specification, either directly as a bare command, or within one of the component's interfaces. If a command or his interface is provided by a module, the module has to define it. The instance parameters of a command are distinct from its regular function parameters.

Below a declaration of a bare command outside an interface is shown.

```

module A {
    provides command void h ();
}
...

```

In addition the *command* keyword can be used in an interface declaration.

```

interface Z {
    command void i ();
}

```

Finally, the *command* keyword is used when the function is actually been defined.

```

...
implementation {
    command void j () {
        //some statements
    }
}

```

The translation to C takes place by simply omitting the keyword, like below.

```

...
void j ();
...

```

**event** The *event* storage class specifier indicates that the function is part of a component's specification, either directly as a bare event, or within one of the component's interfaces. If an event's interface is used by a module, the module has to define it. Contrary, a module has to implement a bare event, if it is provided by a module. The instance parameters of an event are distinct from its regular function parameters. Below a declaration of a bare command outside an interface is shown.

```
module A {
    provides event void h();
}
...
```

In this case A must implement the event *h*. If *h* would have been an event declared inside an interface and A had been provided this interface, then A doesn't have to implement *h*, but the component connected to A has to do it instead.

As *command*, the *event* keyword can be used in an interface declaration.

```
interface Z {
    event void i();
}
```

Finally, the *event* keyword is used when the function is actually been defined.

```
...
implementation {
    event void j() {
        //some statements
    }
}
```

**default** A module can specify a default implementation for a used command or event. A compile-time error occurs if a default implementation is supplied for a provided command or event. Default implementations are executed when the command or event is not connected to any command or event implementation. A default command or event is defined by prefixing a command or event implementation with the *default* declaration specifier, as it can be seen below.

```

...
implementation {
    ...
    default command void c() {
        //some statements
    }
    ...
}

```

Since this has only a sense for the nesC compiler, there is no translation for the C language.

**nx\_struct and nx\_union** External structures and unions are declared like C structures and unions, but using the `nx_struct` and `nx_union` keywords. An external structure can only contain external types as elements. Currently, external structures and unions cannot contain bit-fields.

These two keywords are translated into C by defining them simply as preprocessor macros as shown below.

```

#define nx_struct struct
#define nx_union union

```

**norace** A mechanism that is detecting data races is implemented in nesC, but it is possible that data races that cannot occur in practice, e.g., if all accesses are protected by guards on some other variable, are reported. To avoid redundant messages in this case, the programmer can annotate a variable *v* with the *norace* storage class specifier to eliminate all data race warnings for *v*. The *norace* keyword should be used with caution. Since there is no data race detection mechanism in the C language, this keyword is not translated into C.

## 4.3 Configurations and Wiring

While the modules' implementation consists in allocating state and implementing executable logic, the configurations' implementation consists in the declaration of component elements, declarations, and connections.

A component element specifies the components that are use to build the configuration, a declaration can declare a typedef or tagged type (other C declarations are compile-time errors, like in 4.1) and a connection specifies a wiring statement.



Wiring in nesC means to connect one component to another one, allowing so one module to call functions defined in another one. One such statement can connect two different sets of specification elements:

- The configuration’s specification elements specified in the configuration’s component specification part (see 4.1). These are called external specification elements.
- The specification elements specified in the configuration’s implementation part before the wiring. These are called internal specification elements.

There are three wiring statements in nesC:

- $endpoint_1 = endpoint_2$ : Any connection involving an external specification element. These effectively makes two specification elements equivalent. Let  $S_1$  be the specification element of  $endpoint_1$  and  $S_2$  that of  $endpoint_2$ . One of the following two conditions must hold or a compile-time error occurs:
  - $S_1$  is internal,  $S_2$  is external (or vice-versa) and  $S_1$  and  $S_2$  are both provided or both used,
  - $S_1$  and  $S_2$  are both external and one is provided and the other used.
 This type of connection is also called “Pass Through Wiring”.
- $endpoint_1 \rightarrow endpoint_2$  (link wires): A connection between two internal specification elements. Link wires always connect a used specification element specified by  $endpoint_1$  to a provided one specified by  $endpoint_2$ . If these two conditions do not hold a compile-time error occurs.
- $endpoint_1 \leftarrow endpoint_2$  is equivalent to  $endpoint_1 \rightarrow endpoint_2$ .

To conclude this section some wiring examples are presented.

```

configuration A {
}
implementation {
    components M, O;
    M.X -> O.X;
}

```

In this case, X is an interface defined somewhere. M and O are two modules, where M provides interface X and O uses it. This means that M has to implement the commands contained in X and O has to implement X's events.

Another wiring example, where the = connection operator is used, is shown below.

```
//File A.nc
configuration A {

}
implementation {
    components M, N;
    M.X -> N.X;
}
//End of file A.nc

//File N.nc
configuration N {
    provides interface X;
}
implementation {
    components O;
    X = O.X;
}
//End of File N.nc
```

The = connection operator causes *N.X* to be renamed in *O.X*

The third example shows a passthroug wiring, where two external specification elements are connected with the = operator.

```
//File A.nc
configuration A {

}
implementation {
    components M, N, O;
    M.X -> N.Z;
    N.W -> O.X;
}
//End of file A.nc
```

```

//File N.nc
configuration N {
    provides interface X as Z;
    uses interface X as W;
}
implementation {
    Z = W;
}
//End of File N.nc

```

Surprisingly, all three wiring statements have exactly the same C translation that consists basically in a so called intermediate function for each event and command contained in X, as it can be seen below.

```

//an implementation like the
//following for each
//event contained in X

```

```

O.X.e() {
    M.X.e();
}

```

```

//an implementation like the
//following for each
//command contained in X

```

```

M.X.c() {
    O.X.c();
}

```

### 4.3.1 Implicit Wiring

Typically a wiring statement consists in wiring a used interface used in a module to a provided interface provided in an other module with the following syntax.

$$X.A \rightarrow Y.B,$$

where A and B are two instance-names of the same interface-type, let it be for example type Z, where interface of type Z is used in module X and provided in module Y. The instance-name has not necessary to be the same.

It is possible to use a statement called implicit wiring or implicit connection by omitting the interface in one of the two sides of the connection, for example in the right side, obtaining the following syntax,

$$X.A \rightarrow Y,$$

if exactly one specification element B is found in Y such that

$$X.A \rightarrow Y.B,$$

forms a valid connection. Again, this means that the two instance-names A and B must be of the same interface-type Z and in addition there cannot be more than one used or provided interface of type Z in module Y.

### 4.3.2 Parameterized Wiring

Sometimes, a component wants to provide many instances of an interface. To simplify this purpose parameterized interfaces were introduced. An interface declaration without instance-parameters (e.g., interface X as Y) declares a single interface to this component. A declaration with instance-parameters (e.g., interface A[uint8\_t id]) declares a parameterized interface, corresponding to multiple interfaces to this component, one for each distinct tuple of parameter values. In the same way interface A as S[uint8\_t id1, uint8\_t id2] declares 256 \* 256 interfaces of type A. The types of the parameters must be integral types, enums are not allowed at this time.

An example of how to use a parameterized interface is shown below.

```

module A {
    provides interface X[int id , char d];
}
implementation {
    command int X.c[int id , char d]() {
        //some statements
    }
}

```

X is a normal interface <sup>2</sup> with commands and events declared in it. Each command is defined like in the example with the same parameters in square brackets as in the specification element part of the component. This leads to the following C translation.

---

<sup>2</sup>Parameterized interfaces should not be confused with typed interfaces, which are a nesC v1.2 construct and completely different.

```
int X.c(uint8_t id, unsigned char d) {  
    //some statements  
}
```

As it can be seen, the parameters become simple function parameters in the C language.



# Chapter 5

## General compiler design

Translators take an input language and apply transformation rules in order to generate corresponding output in another language. Compiler differ from translator only in the output language they generate. While translator generally output some kind of high-level language, compiler output assembly language or straight machine code <sup>1</sup>.

The construction of a compiler can be divided in three different phases <sup>2</sup>: front end, middle end and back end. The middle-end phase is not used anymore nowadays. A typical compiler architecture is shown in figure 5.1.

This chapter describes these phases and introduces a tool, JavaCC, used for the construction of some parts of the translator.

### 5.1 Front end

The front end takes an input and analyzes the source code to build an internal representation of the program, called the intermediate representation. It also manages the symbol table, if it is present, a data structure mapping each symbol in the source code to associated information such as location, type and scope.

This is done over several phases, which includes the following:

1. Line reconstruction

Languages which allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a

---

<sup>1</sup>see also section 1.4 for a detailed compiler and translator definition

<sup>2</sup>There are also one-pass compiler that have only one phase, but are not described here

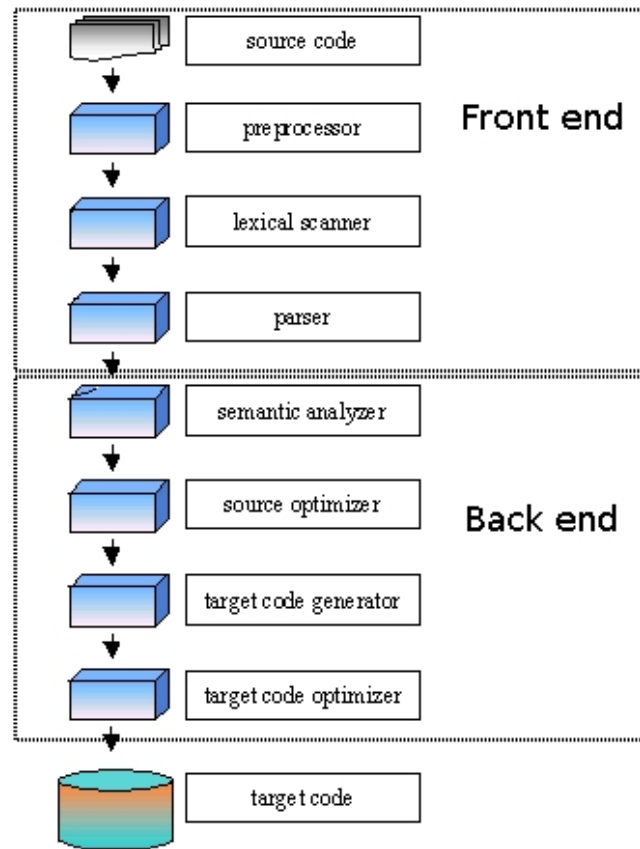


Figure 5.1: A typical compiler architecture

canonical form ready for the parser. This phase is not necessary in the present project, because nesC doesn't allow such keywords.

## 2. Lexical analysis

This phase breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer, lexer or scanner.

## 3. Preprocessing



Some languages, like C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically, the preprocessing phase occurs before syntactic or semantic analysis.

#### 4. Syntax analysis

This phase involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

#### 5. Semantic analysis

During this phase the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

## 5.2 Back end

The back end takes the intermediate representation and generates the final output of the translator or compiler. This is also done over several phases.

#### 1. Analysis

This phase consists in the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis and escape analysis. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.

#### 2. Optimization

The intermediate language representation is transformed into functionally equivalent but faster or smaller <sup>3</sup> forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation or even automatic parallelization. To speed up the development process this phase is not used in the present project and can maybe added in a later version of the translator.

### 3. Code generation

The transformed intermediate language is translated into the output language.

## 5.3 JavaCC - a parser generator

To realize the front end of the parser apart from the semantic analysis, the JavaCC ([JCCa]) parser generator has been used in the present project, which creates a top down parser (recursive descending) <sup>4</sup>.

In JavaCC the lexical specification in form of regular expressions and the grammar rules, called also grammar productions, are written down in the same file. Another important feature of JavaCC is the use of EBNF (Extended Backus Naur Form), which enriches the BNF (Backus Naur Form) with regular expressions. JavaCC produces LL(1) <sup>5</sup> parsers by default, but is not limited to  $k=1$  so  $k>1$  is possible at any choice position. This functionality is achieved with syntactic and semantic lookahead <sup>6</sup>. A JavaCC-generated parser can thus be LL( $k$ ) at the points where it is needed allowing so to parse all LL class of grammars.

Once all the grammar rules and lexical specification are written in a *jj* file, this file can be compiled by JavaCC and the Java classes that implement the compiler are generated as it can be seen in figure 5.2

## 5.4 JJtree - a parse tree generator

In the present project JJtree is used for building the parse tree in conjunction with the visitor design pattern. In this case the grammar written in a *jjt* file is used by JJtree to generate a *jj* file containing the grammar with generated

---

<sup>3</sup>in the sense that they need less memory

<sup>4</sup>see [ASU86] for the meanings of recursive descending and top down vs bottom up parser

<sup>5</sup>see [ASU86] for more details

<sup>6</sup>see [JCCa] for a complete description of the lookahead construct

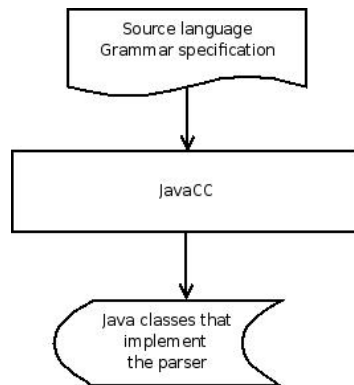


Figure 5.2: File generation process of JavaCC

semantic actions for building the parse tree. In addition the visitor interface is generated allowing the separation of the structure and algorithm classes.

The implementation of the compiler resides only in the visitor classes. Therefore the semantic classes are compact and only contain the required visitor code and the compiler is not spread over dozens of semantic classes. Each semantic class is implemented by a Java class, which represents a node of the parse tree. Consequently, JJtree creates a class hierarchy which implements the Node interface. Part of the class hierarchy generated by JJtree for the present project can be seen in figure 5.3, where two classes, AST-Expression and ASTNesCFile, are represented. If a new node needs to be represented in the parse tree the corresponding Java class is automatically derived from SimpleNode.

Different types of concrete visitors can be implemented from the generated visitor interface, like for example a visitor responsible for the code generation. This is a flexible solution, because the actual semantics are done in separate visitor files, what gives a clear and scalable design.

JJTree generates semantic classes for all non terminals in a given grammar, which automatically get instantiated within the semantic actions. The following options written in the jjt file generate a parse tree in conjunction with the visitor pattern.

```

options {
    MULTI = true;
    VISITOR = true;
}
  
```

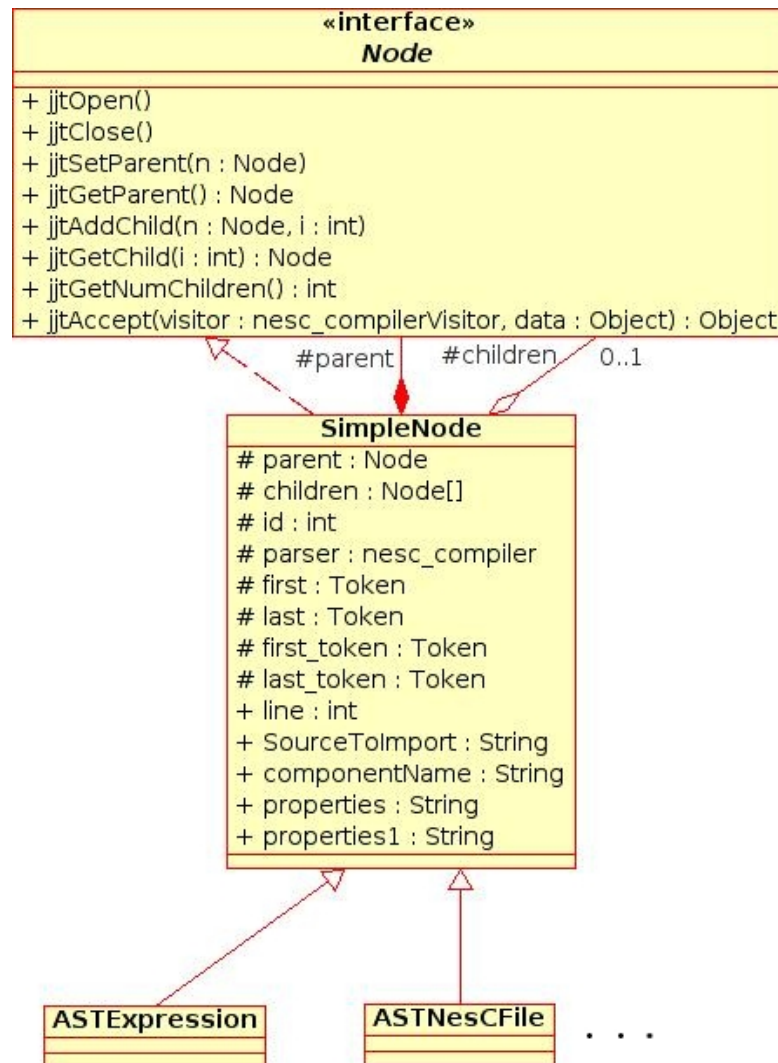


Figure 5.3: Class hierarchy representing the parse tree

# Chapter 6

## Architecture

### 6.1 Introduction

### 6.2 Software Requirements specification

#### 6.2.1 Users of the software

In the following the type of users that are most likely going to use the software is listed.

1. Porter

Programmer that wants to port an existing TinyOS application to the Erika operating system.

2. Coder

Programmer that codes a nesC application for the Erika operating system from scratch.

3. Maintainer

Programmer that extends the nesC-Erika source code and fixes possible bugs.

4. Installer

User that installs the nesC-Erika software on a PC.

5. Tester

User that tests the correctness of the nesC-Erika software.

### 6.2.2 Use cases

In this section a brief summary of the use cases is presented.

1. Porter

The Porter compiles an existing TinyOS application for Erika by simply doing copy paste of the existing code. The porter should be able to do some manual changes on the generated files to optimize the application. He should get a final executable ready to be copied on the target platform.

2. Coder

The coder should be able to code a new application for the Erika operating system using the nesC language. He should obtain a final executable for his desired target platform ready to be copied on the target platform.

3. Maintainer

The maintainer needs documented source code to understand the code that has been previously written. It is advisable that the maintainer finds comments in the source code and that he writes comment in the new source code that it is been produced. The maintainer should be able to log software changes that he is going to do and to view previously changes done in the source code.

4. Installer

The installer should be able to install nesC-Erika on a PC by simply executing an installation script. Installation instructions and software requirements should be available for the installation.

5. Tester

NesC source codes samples, which tests some specific language constructions should be available for a testing of new added nesC-Erika source code. In addition TinyOS demo applications should be available to make final validation tests.

### 6.2.3 Requirements definition

Based on the different use cases the following user requirements are defined.

1. Translation of a nesC-TinyOS application to a C application for Erika.
2. Extension of the nesC language to support Erika specific constructs.
3. Logging of software changes
4. Testing the correctness of existing and future source code.

### 6.2.4 System requirements specification

In this section functional and non functional system requirements are specified. Functional system requirements are those requirements that the system has to offer to satisfy the needs of the different types of users. Non functional are those requirements that the system has to meet to work correctly.

#### Functional system requirements specification

1. Translation of a nesC-TinyOS application to a C application for Erika.
  - 1.1 The user should be able to specify the source files as command line input.
  - 1.2 A C file needs to be generated.
    - 1.2.1 The C file must be compilable by the target compiler, which generates the final executable.
  - 1.3 An OIL file needs to be generated.
    - 1.3.1 The OIL file must be compilable by RT-Druid, an Erika specific tool.
  - 1.4 The output files should be generated in a human readable way to permit manual manipulation.
  - 1.5 Any programming error should be reported in a human readable way.
    - 1.5.1 The line number of the error should be signaled
    - 1.5.2 The column number of the error should be signaled.
2. Extension of the nesC language to support Erika specific constructs.
  - 2.1 A nesC API for Erika should be offered to permit coding of Erika applications in the nesC programming language.

- 2.2 For supporting Erika features that are not available in TinyOS additional nesC keywords and constructs are needed.
- 3. Logging of software changes
  - 3.1 Old versions of the software must remain available.
  - 3.2 Changes in the software must be tracked.
- 4. Testing the correctness of existing and future source code.
  - 4.1 The written source code should be tested against existing TinyOS applications.
  - 4.2 To permit a faster development process a set of simple and short applications is needed to test specific language constructions.
    - 4.2.1 The set should be increased with each new added language construct.

### **Non functional requirements specification**

- 1. Translation of a nesC-TinyOS application to a C application for Erika.
  - 1.1 The system should offer a tool, which takes nesC source files as input and generates as output a C file, an OIL file and a final executable.
  - 1.2 The system should be able to parse nesC code. If some errors in the source code are detected the compilation process stops and the errors are reported.
- 2. Extension of the nesC language to support Erika specific constructs.
  - 2.1 The nesC API for Erika should contain Erika primitives.
  - 2.2 The names chosen for the nesC components should reflect the existing TinyOS nomenclature to permit the translation of existing nesC-TinyOS applications.
  - 2.3 Erika has some features that are not present in the TinyOS operating system. The system has to offer a mechanism for supporting those features.
- 3. Logging of software changes
  - 3.1 The system has to offer a mechanism, where old versions of the software are automatically backed up, after a new version is stored.



- 3.2 The system has to offer a mechanism, which makes it possible to log software changes.
4. Testing the correctness of existing and future source code.
  - 4.1 The system should contain a set of TinyOS demo application for testing the correctness of each component.
  - 4.2 The system should contain a set of nesC construct specific applications to test the correctness of the system.
    - 4.2.1 A new application with the desired new nesC extension should be added before the actual extension of the source code.

## 6.3 System Design

To simplify the realization of the system, the project is split as first instance in different activities. Further one each activity is split in different components and tasks.

### 6.3.1 Activities

From the system requirements it is possible to divide the whole project in the following activities.

- Realization of a nesC to C translator.
- Realization of a nesC library for Erika.
- Installation of a version control and backup mechanism.
- Preparing of the testing software.
- Selecting of a cross compiler for target platforms.

The rudimental Gantt chart shown in figure 6.1 shows the temporal relationships between the different activities.

Some of the activities require the writing of source code, whereas other activities simply require to find some existing tools that can be used for the system realization. In the following subsections a detailed description of the tasks needed to complete each activity will be given.

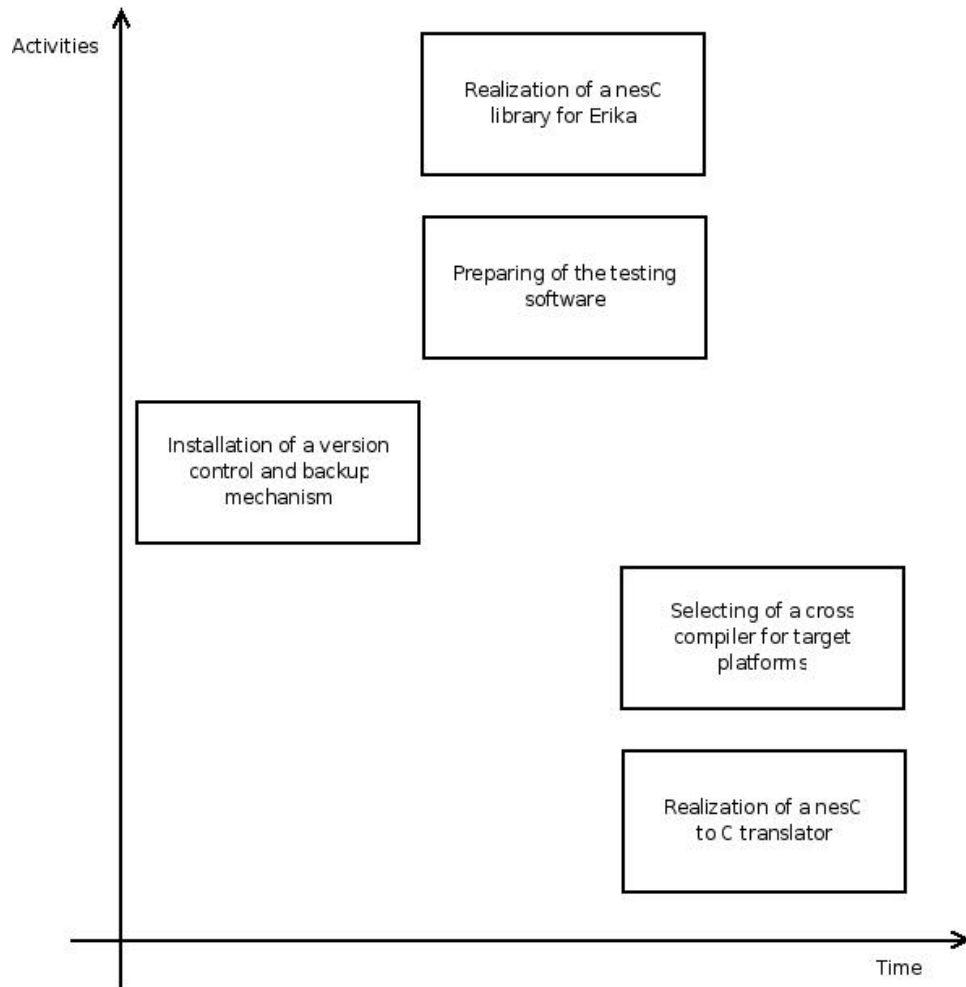


Figure 6.1: Gantt chart of the different activities

### 6.3.2 Realization of a nesC to C translator

This activity is the core activity of the whole project. Its main purpose is to realize a translator able to compile nesC code to C code. In the early stages of the project it should also be able to translate TinyOS specific macros and function calls to Erika macros and function calls, because at this time a nesC library for Erika is not available. This can be done by specifying some header files in which appropriate defines will be written.

The translator itself can be divided in the following components.

1. A front end <sup>1</sup> that has the purpose to gather command line options and

---

<sup>1</sup>Not to confuse with the compiling design phase explained in chapter 5

to specify directories in which it is possible to find header files.

2. The parser that is able to parse both nesC and C code.
3. The Error checker, which has the responsibility to check if there are semantic errors in the source code, which are not handled by the parser.
4. The C code generator that is responsible for generating the C files with the appropriate code.
5. The OIL code generator, which generates the OIL file.

The following diagram gives an overview of the whole translation process.

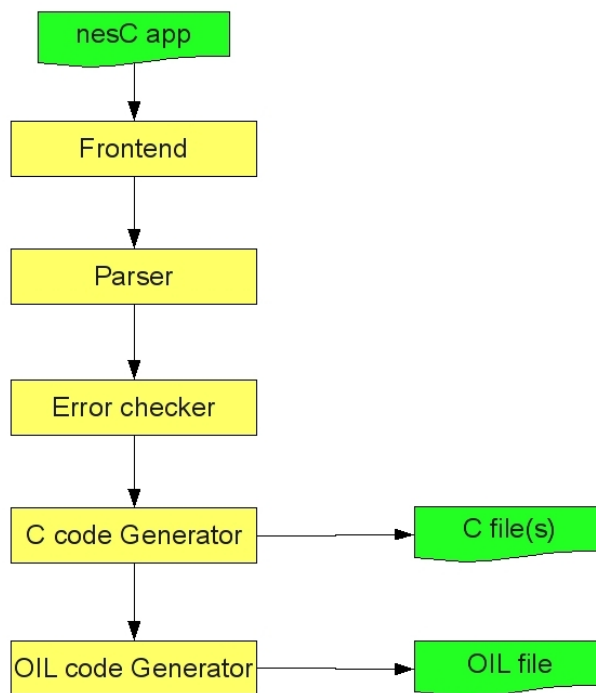


Figure 6.2: Translation process

In the following subsections a detailed description of each translator's component is given.

### **The front end**

This component should be able to gather command line options, to take the input files and pass them to the right components and to search for specified header files in the filesystem. In addition it should also be able to call all the different software tools of which the compiler is composed. The command line options are analyzed and an underlying component (i.e. the cross-compiler) that is able to handle the option is searched, if the front end itself doesn't recognize the option. To implement it, the front end of the existing nesC-TinyOS compiler can be used as base. Instead of calling the nesC-TinyOS compiler it should be adapted to call first the nesC-Erika compiler and then the right cross compiler for the target platform. The system design of the front end can be seen in figure 6.3.

### **The parser**

The parser parses nesC source files and outputs a syntax tree. It must be able to recognize both the nesC and C grammar, because nesC source files contain pure C code as well. The ANSI C grammar is not enough, because some gcc extensions are used. Therefore the parser has to be able to recognize gcc specific constructs too. To realize the parser, it has been decided to use the JavaCC [JCCa] parser generator, which generates appropriate Java source code that implements a parser given a BNF grammar. This means that the parser is platform independent as long as a Java Virtual machine is available for the platform on which the parser should be installed. Additionally also JJTree, a tool included in the JavaCC package, is used for building the syntax tree. Once the grammar of the language is given as input the generated parser is automatically able to handle syntactic errors.

The most difficult part of the parser writing task is to cope with C's typedefs and C's preprocessor statements. Because the C grammar is not a context free grammar, the typedefs need to be stored in a table to allow a correct parsing of C statements. To handle the C's preprocessor statements three solutions will be discussed in the following, but the choice of which solution to use will be left to the implementation phase.

A first solution could consist in preprocessing the source files with an external tool, like the gcc preprocessor. This leads to the clear advantage of saving development time. In this case every time a new file needs to be parsed, it is first preprocessed and the preprocessed output is then given as input to the nesC-Erika compiler.

Another solution could be to code a C preprocessor and to integrate it directly in the parser, but this is not an easy task, because C's formal

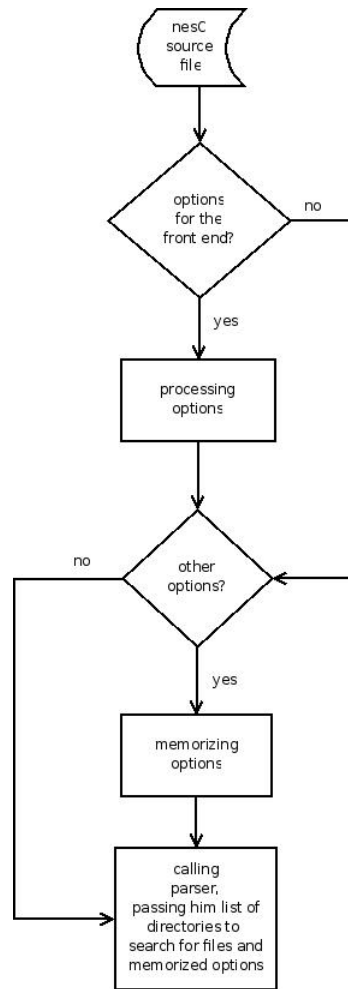


Figure 6.3: Front end system design

preprocessor grammar is not suitable for mechanical parsing. If this solution would be used, then it is also needed to decide whether to construct a formal grammar that is suitable to be given as input to a parser generator or to code a preprocessor without the help of a parser generator.

The last solution is to skip all C preprocessor statements and to leave them as they are in the generated C file. The generated C file is then anyway preprocessed by the cross compiler, so preprocessing statements are finally expanded. This is of course the simplest solution, but its drawback is that header files wouldn't be parsed and so some information (i.e. typedef statements) are lost. To avoid this it is possible to parse only the *"#include"* statements and to ignore all the remaining statements. Also in this case

the problem of symbols defined by the *"#define"* statement remains. If this statement is not parsed there is no chance to store those symbols and this makes it impossible to parse C files that use symbols defined by the *"#define"* statement correctly. So this solution would require a workaround in the implementation phase to solve the problem.

As said before, if the source files respect the nesC syntax, the parser finally creates a syntax tree, which can then be used by other compiler components for other tasks. The system level design of the nesC parser is illustrated in figure 6.4.

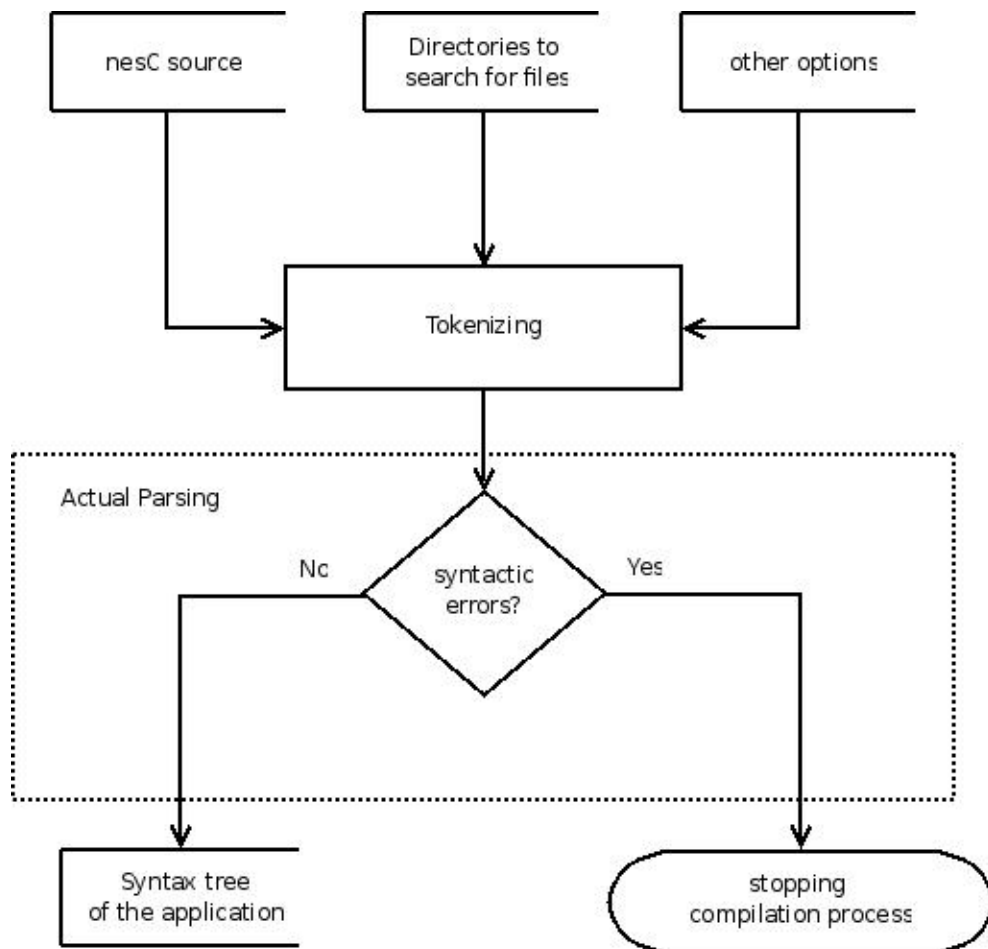


Figure 6.4: Parser system design

### **The Error checker**

The purpose of the error checker is to handle those errors that are not handled by the parser. This includes all non syntactic errors, i.e. semantic ones. To speed up the realization process in the early stages of the project this task is done by the existing nesC-TinyOS compiler as specified better in section 6.4. This permits to concentrate on other implementation tasks and to obtain a first working compiler version in a shorter time.

Nevertheless some characteristics are given in this section to build an error checker in the future. It is important that the error checker is able to recognize all the nesC semantic errors, whereas it is not mandatory that it is able to recognize C semantic errors, because those are handled in a later stage by the target cross compiler. The error checker will consist basically of three components: a tool to fill the data structures where the information of a nesC program are stored, an another one that has to check if the informations stored are consistent with the semantic of the nesC programming language and additionally it has to store any errors found in the source that is going to be compiled. The last component is responsible for reporting the errors to the programmer.

The first component consists of "table like" data structures, where the relevant nesC data is stored. If the error checker is for example written in the Java language ArrayLists, Vectors or HashTables can be used for the implementation of those data structures . The second component traverses the syntax tree and checks if the informations found there are consistent with the informations stored by the first component. The last component prints the errors stored by the previous component to the programmer. The system level design in figure 6.5 summarizes the functionality of the error checker.

### **The C code generator**

The C code generator has the duty to generate C code by visiting the syntax tree that was generated by the parser. Figure 6.6 shows the principles of this process at the system level.

One of the main problems by the construction of a C code generator is that once something has been written on the output files it cannot be changed anymore. This is for a language as nesC a problem, because the C translation of some nesC statements found in a file depend on some statements that can be found in another files. It is possible that this fact is repeated recursively for an undefined number of times and in addition it is not known the order with which those files are going to be compiled. Therefore it is for those statements necessary that the C code generator is able to access

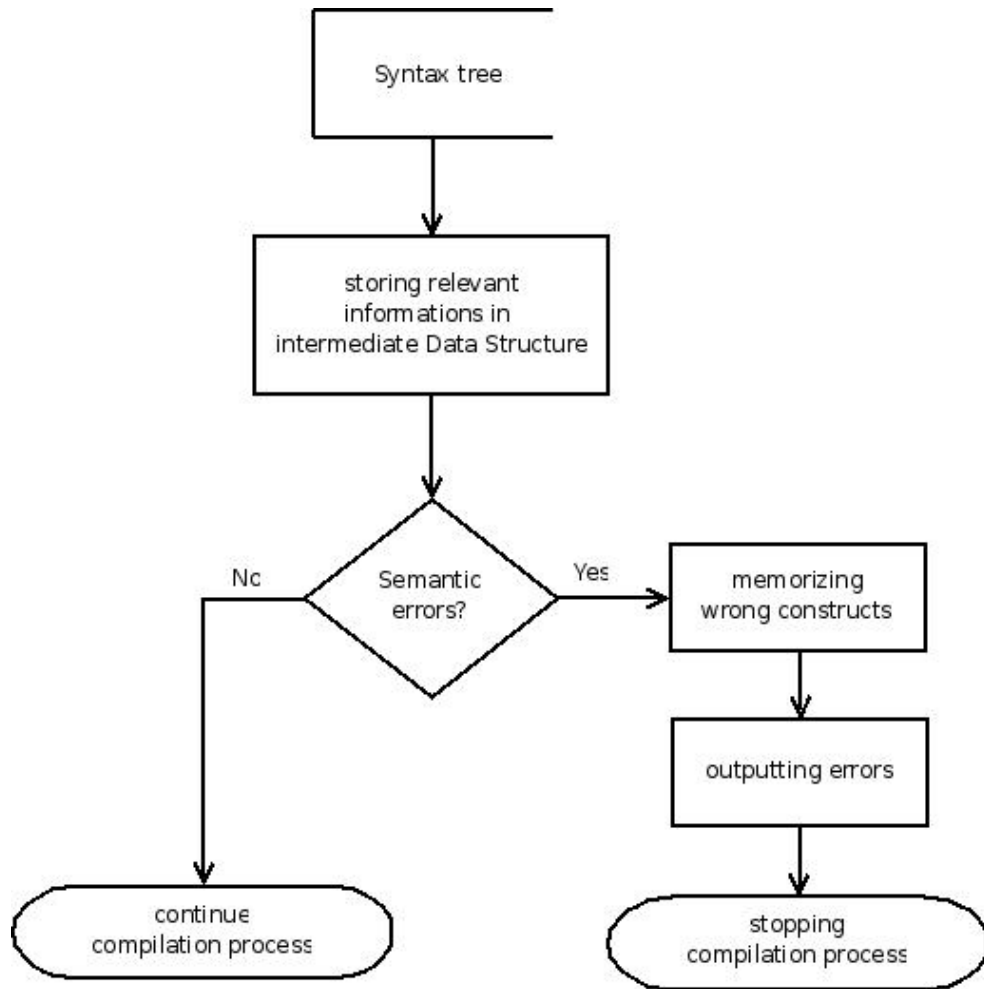


Figure 6.5: Error checker system design

a data structure, where the relevant informations are stored in advance. In the following the most important informations that need to be stored in an intermediate representation are listed.

- All connections present in all configuration files.
- A list containing all used components.
- A list containing all provided components.

This data structures must be constructed before the actual code generation phase, but after the parsing phase. Some data structures build during the error check phase can be reused for that purpose.



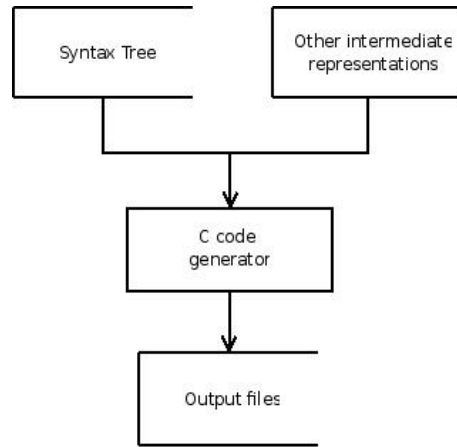


Figure 6.6: C code generator system design

### The OIL code generator

The OIL code generator's job consists in generating an OIL file by visiting the syntax tree generated after the parsing phase. Again, some intermediate data structure, which contain informations like the list of tasks or the list of interrupt handlers, are useful to support this duty. The system level design of the OIL code generator is shown in figure 6.7. If a TinyOS program is being translated some decisions in the generation are straightforward, i.e. the scheduler choice, because in TinyOS only one scheduler is available unlike in Erika, where more schedulers are available. If a new program for Erika should be coded from scratch using the nesC language and if it is wanted to set some OIL options from the nesC language level a mapping from nesC to OIL is necessary. This is beyond the aim of this project and not described here further.

#### 6.3.3 Realization of a nesC library for Erika

A nesC library for Erika is needed to have a fully working nesC-Erika environment. To permit a proper translation of existing TinyOS applications the components contained in the Erika library should have the same file names as the existing TinyOS components. The library should hide all the C code of the Erika operating system behind nesC components allowing the programmer to code in nesC language only. This would also allow an automatic translation of TinyOS system call to Erika system calls.

Because this task requires a long development time, to speed up the development process in the beginning a set of header files is instead used.

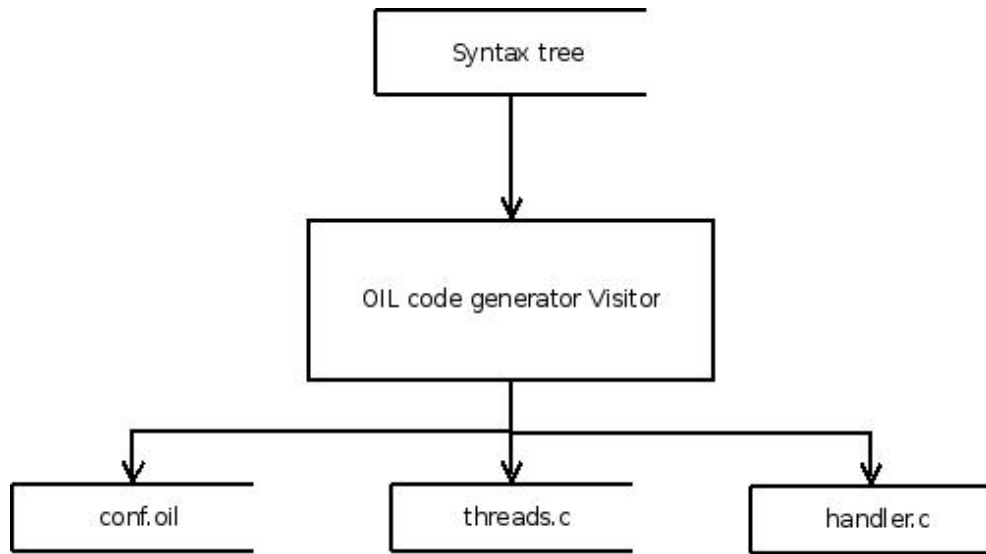


Figure 6.7: OIL code generator system design

The header files have the purpose to map TinyOS specific C code, like system calls, to Erika code. These files are included always as *"include"* statements at the beginning of the generated main C file. This set of header files is used as temporary substitution of a real nesC-Erika library.

### 6.3.4 Version control

An automatised version control system is needed to backup the changes done in the source code and to meet the requirements of maintainability (see non functional requirements 3.1 and 3.2). As version control system it has been chosen RCS [RCSa], because it is the most suitable for single user utilization and the one that requires the smallest configuration effort. In any case it is quite simple to substitute it with the CVS version control system, if at some point more people should work on this project.

### 6.3.5 Testing software

Two different types of compiler tests are done, the first one consists in testing the compiler's behaviour only on a software basis. This type of tests, given an application that compiles with the original nesC-TinyOS compiler checks, if the application compiles with the nesC-Erika compiler as well. Similarly, if the nesC-TinyOS compiler gives an error, also the nesC-Erika compiler

should give an error, or at least it should stop the compiling process. The other type of tests is a hardware based type of tests and consists in testing an existing TinyOS application compiled by the nesC-Erika compiler on a real hardware platform. The design of the software based tests are described in section 6.6 and since the hardware tests are some kind of final validation tests, they are described towards the end of the document in chapter 8.

### 6.3.6 Selecting a cross compiler for target platforms

This is the final activity that needs to be done for the realization of the project. For every microcontroller on which nesc-Erika software should be executed a different compiler is needed. For that reason the gcc compiler collection is the best choice, because at the time of writing this paper all microcontrollers that are supported by Erika are also supported by gcc.

## 6.4 Software design

In this section the design of the software that is going to be realized is described with the help of data flow diagrams. For timing issues some simplifications are done in comparison with the plannings made during the system design phase. In particular on the error checker and on the nesC library some major simplifications are done to assure a faster development time.

Figure 6.8 gives an overview of the whole compilation process. As it can be seen there, a nesC application is compiled by first checking if there are some errors with the help of the nesC-TinyOS compiler. If there are some, the compilation process stops, otherwise an XML file is generated, which is the input in addition to the nesC application to nesC-Erika. NesC-Erika generates then some C files and an OIL file that are finally compiled to an executable by the RT-Druid tool.

### 6.4.1 nesC to C translator design

The design of this software component is described as the sum of the design of the software submodules, as explained in section 6.3.1. The simplification changes done on some modules like the error checker and the nesC-Erika library have an effect on some other software modules as well.

#### Design of the Front end

The front end is made of one software modules. It consists in a shell script that calls first the original nesC compiler. This is done to obtain an optimal

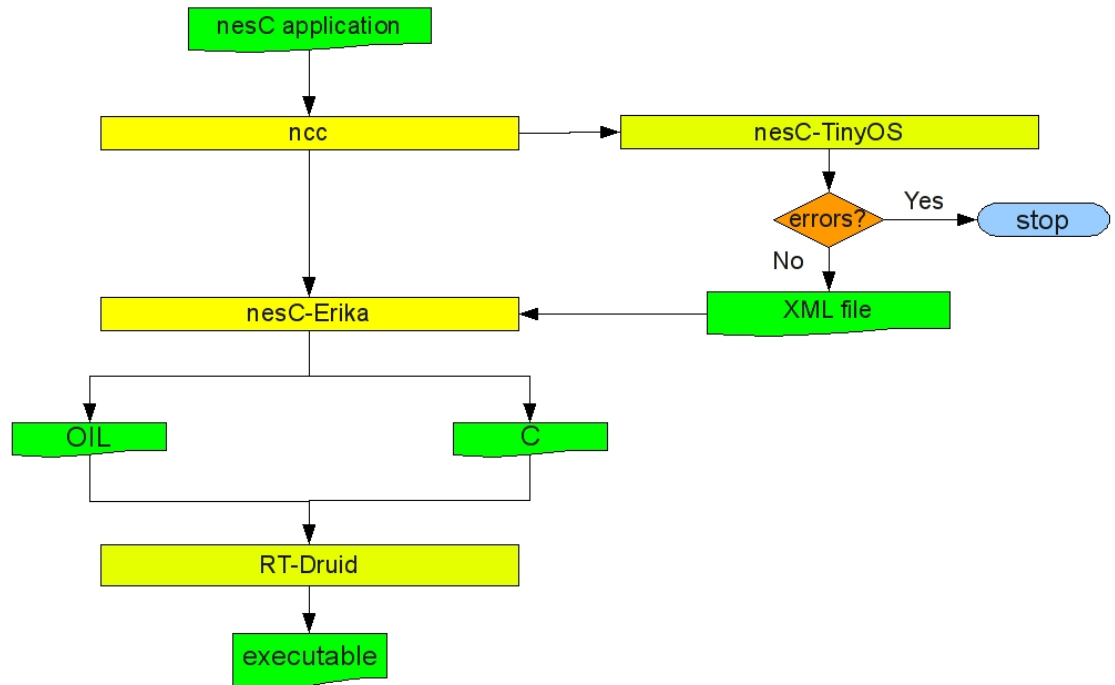


Figure 6.8: Compilation process

error checking and an intermediate data structure, where useful information are stored as explained in later sections. After that the modified original nesC Perl front end is called. This component will then call the actual nesC-Erika compiler, which does the rest of the work. Figure 6.9 shows the front end's software design.

### Parser design

The Parser is obtained by inserting the nesC 1.1 grammar rules found in [GLCB03] as well as the nesC 1.2 grammar rules found in [GLCB05]. The reason for inserting both set of rules is that otherwise applications written with the 1.1 syntax, like all TinyOS 1.x applications, are not compilable, if only the 1.2 set of rules is inserted. Of course if only the 1.1 set of rules is inserted, applications using the 1.2 syntax, like all TinyOS 2.x applications cannot be parsed.

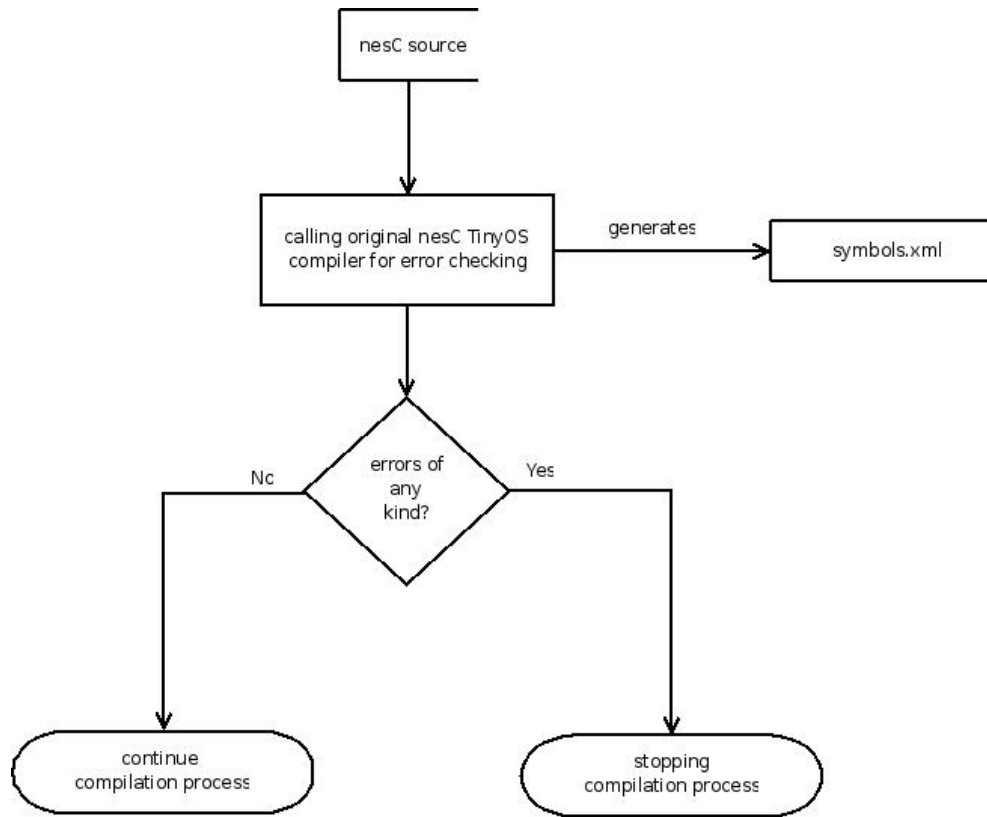


Figure 6.9: Front end software design

After an application is successfully parsed the parser generates a syntax tree. For that reason the JavaCC preprocessor, JJTree, for automatic building of a syntax tree is used. Therefore the grammar rules are inserted in a .jjt file. After the JJTree preprocessor is run on it a jj file will be generated.

Finally the actual parser generator JavaCC is run and some Java classes are generated, along with all the classes that form the syntax tree. Below the classes that are not part of the syntax tree are listed.

The syntax tree consists in a class for every node of the tree that is derived from a BaseNode class, which implements the Node interface as depicted in diagram 5.3.

A node of the syntax tree represents a grammar rule. Not for all grammar rules it is necessary to have a node in the syntax tree. For example the *CallKind* grammar rule needs one and the *AdditiveExpression* grammar rule does not. The decision if a grammar rule needs or doesn't need a node in the syntax tree is left from case to case to the implementation phase and it

depends on the fact if a transformation is needed to be done or if it is enough just to leave the token as it is in the output file. For example a "+" sign has the same translation in nesC code as well as in C code and therefore the AdditiveExpression rule does not need a node in the syntax tree. Instead, the CallKind rule has a semantic in the nesC language only and therefore a node is necessary to make some changes on it.

### **Error checker**

To speed up the development process in the first compiler versions the error checker's work will be done by the existing nesC-TinyOS compiler. As explained in 6.4.1 and shown in figure 6.9 the original nesC-TinyOS compiler is called by the front end. If an error is detected the whole compilation process stops and the errors are reported. If no errors are found the actual nesC-Erika compiler is executed and the compilation process continues.

### **C code generator**

Also for these software module some simplifications are required to permit a first compiler release in a reasonable time. The necessary intermediate data structures as explained in section 6.3.2 are generated by the original nesC-TinyOS compiler in form of an XML file. After that this file is processed by a SAX parser, which creates some Java helper object that contain information required to generate correctly the C output files. In later compiler versions instead of parsing the XML file it is possible to gather those informations in other ways from the input files. In any case it is really useful to use the XML file in the beginning, because this makes it possible to construct a consistent class hierarchy to contain the information needed to generate the C output. Otherwise it would not be easy to guess, which information should be stored, which not and in which way. The C code generator's software design is summarized in figure 6.10.

### **OIL code generator**

The purpose of this component is to generate an OIL file and two C files. In one of the two C files the interrupt handlers to be used in Erika are defined, in the other one the threads of which the application is composed are declared. The ideal way to implement this component would it be to divide it in two software modules. The first one should store all the tasks and interrupt handlers found in the source files in a data structure, the second one should generate on the base of this informations correctly the OIL file. Again, to speed up the development process, in the beginning only one component is

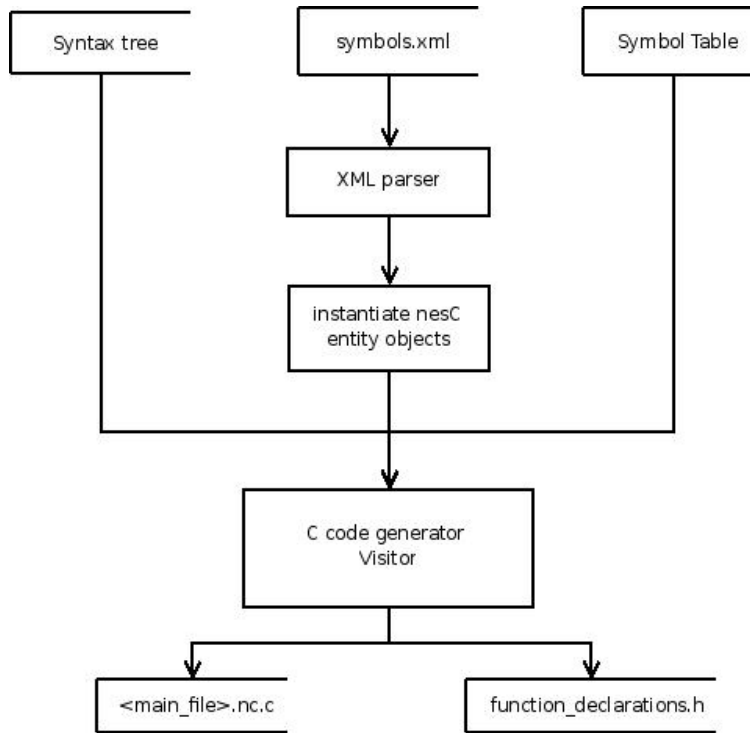


Figure 6.10: C code generator software design

implemented that will do the work of collecting all the tasks and interrupt handlers as well as the work of generating the actual OIL code. Figure 6.11 summarizes the OIL code generator's design.

### 6.4.2 Realization of a nesC library for Erika

As described in section 6.3.3 in the early compiler versions a set of header files is used instead of a real nesC library for Erika. The header files should translate TinyOS system calls to Erika system calls with the use of *#define* statements. Because some system calls available for TinyOS don't have a correspondent Erika system call, those system calls are translated like in the following sample.

In addition to these header file another header files are needed, where TinyOS types are defined, like for example the *bool* type.

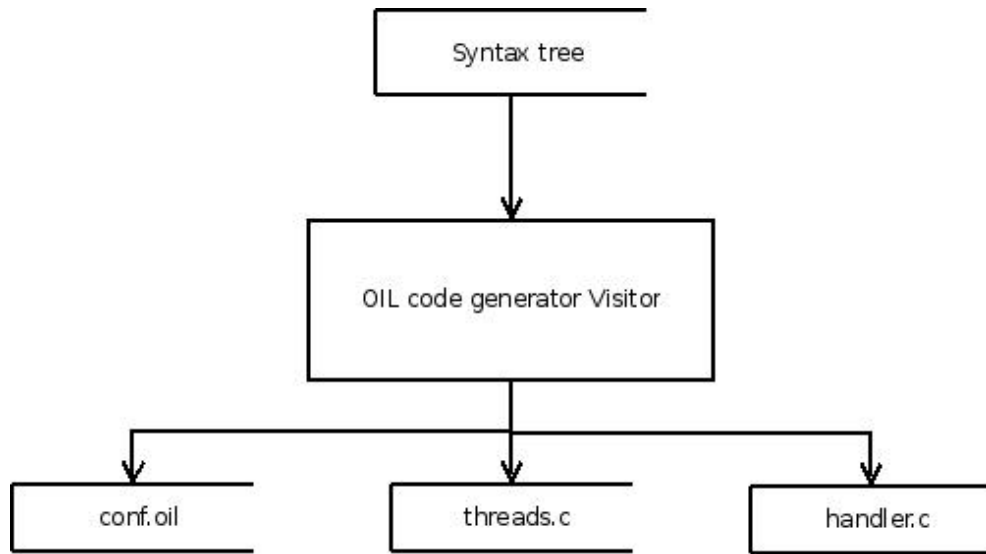


Figure 6.11: OIL code generator software design

## 6.5 Implementation details

Of the software modules identified in section 6.4 the following will be implemented during this project.

- Front end
- Parser
- Symbol Table
- XML parsers
- C code generator
- nesC entity objects
- OIL code generator

In the following sections implementation details and in particular class diagrams of those software modules that are implemented by a class hierarchy are shown.



### 6.5.1 The front end

As explained in the software design phase this module consists in a Perl script. The base for this Perl script is taken from the original nesC-TinyOS package, which is called `ncc`. The same name has been kept also for the correspondent nesC-Erika script which can be found in the nesC-Erika installation directory.

### 6.5.2 The parser

The parser consists mainly of a `jjt` files which contains the lexer and parser rules. This `jjt` file is run through the `jjt` preprocessor, which then generates the `jj` file with the annotations for the generation of the parse tree. This file is then run through the actual JavaCC compiler, which then generates the Java code consisting in a set of classes that implement the parser.

The `jjt` file can be divided in two different parts. One consists in the definition of the tokens in terms of regular expressions that are used by the lexer. Some examples of token definitions are listed below.

An other part consists in the grammar rules written using the EBNF (Extended Backus-Naur Form) syntax. This grammar rules, also called EBNF production rules can contain semantic actions consisting in pure Java code written in it that are executed when a production is recognized. Again, some examples are shown below.

### 6.5.3 The C and OIL code generators

The `C_CodeGeneratorVisitor` as well as the `OIL` code generator classes are derived from the `UnparseVisitor` class, which is a class that implements things that all code generators have in common, like for example methods to print tokens to an output file. This class, on the other hand is derived from the `FirstVisitor` class, which implements the `nesc_compilerVisitor` interface. This class hierarchy is shown in figure 6.12.

### 6.5.4 The symbol table

This software module is implemented by a class hierarchy consisting in the `Scope` class responsible for taking track of scope changes in the source code, a `Symbol` class that maps lexemes to a `HashTable` for faster computation and the `SymbolTable` itself that memorizes relevant symbols found in the source code. This hierarchy is shown in figure 6.13.

The `SymbolTable` is constructed by the `SymbolTableVisitor` class, which is part of the hierarchy shown in figure 6.12.

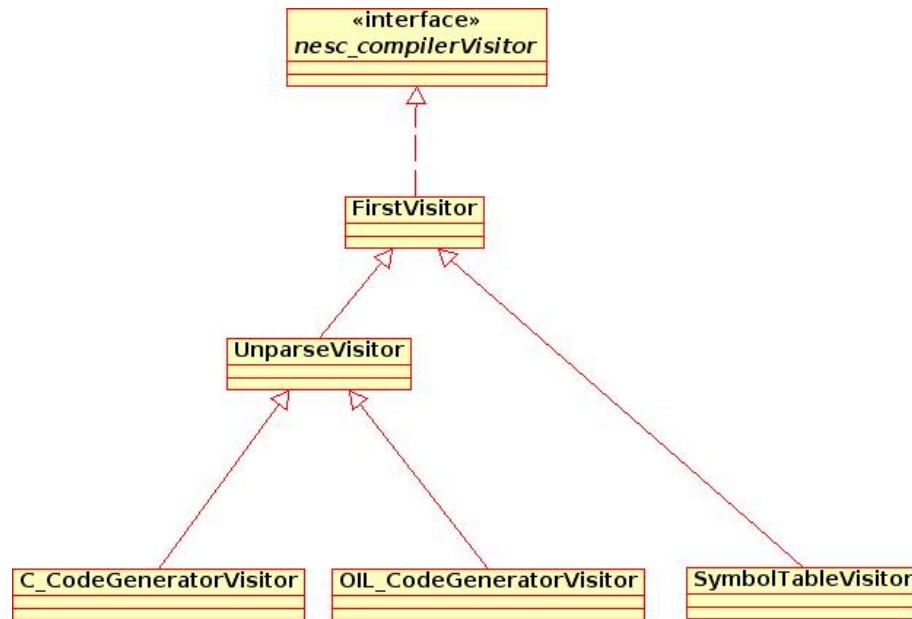


Figure 6.12: The visitors class hierarchy

### 6.5.5 XML parsers

This software module consists in a set of SAX parsers that are gathering some kind of informations that are available in the *symbols.xml* file generated by the nesC-TinyOS compiler. For each type of information that needs to be stored a handler is derived from the FirstHandler class, which on the other hand implements the standard ContentHandler SAX interface. All the objects created by parsing the *symbols.xml* file are stored in the XMLpicker class.

### 6.5.6 The nesC entity Objects

This set of objects are implemented using a class hierarchy. All concrete objects are derived from the abstract class NESCEntity. This objects contain informations like interface instances available in the nesC application. To explain the relationships of the classes one can imagine that for example a NESCEntityInterfaceInstance is composed of several NESCEntityFunctions and each NESCEntityFunction can have zero or more NESCEntityParameters. The class diagram is shown in figure 6.15.

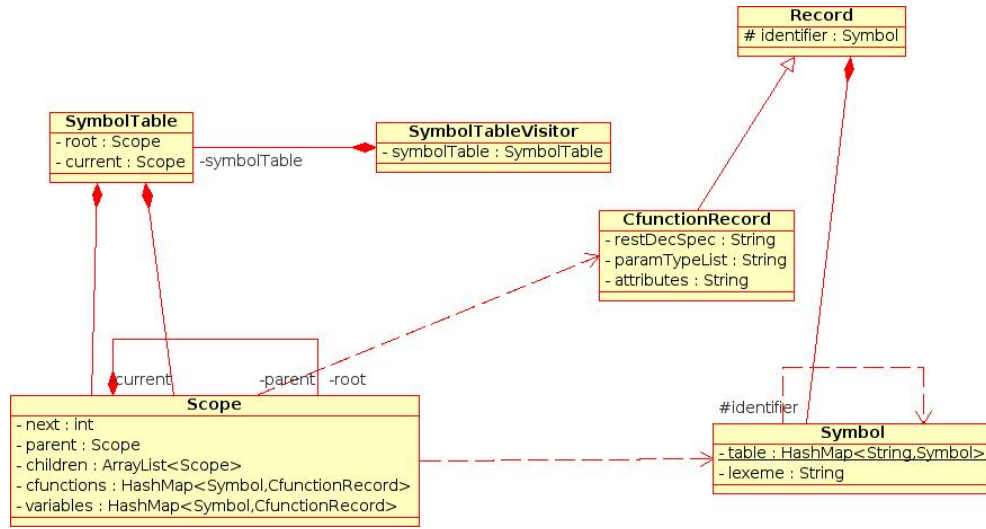


Figure 6.13: Symbol table class diagram

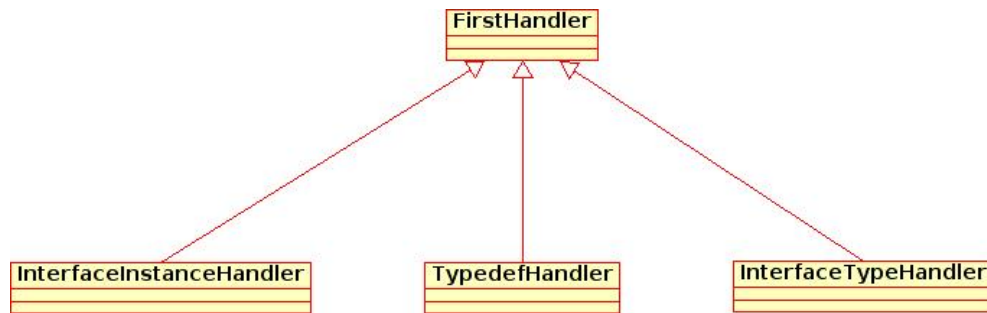


Figure 6.14: The SAX handler hierarchy

## 6.6 Preparing of the testing software

Two separate sets of nesC applications are needed to test the correctness of the translator. The A set contains existing TinyOS demo applications, whereas the B set contains shorter code to test specific nesC language constructions. Another characteristic of the applications contained in the B set is that they are unrelated to the TinyOS environment and therefore they can be used to test the exactness of this compiler's aspect. The nesC components written to implement the Erika library (see section 6.3.3) could be used in the B set as well, because they form nesC testing code that is unrelated to the TinyOS environment, although they may be not so short, but exactly for this reason they can be useful to do final validating tests for the use of the

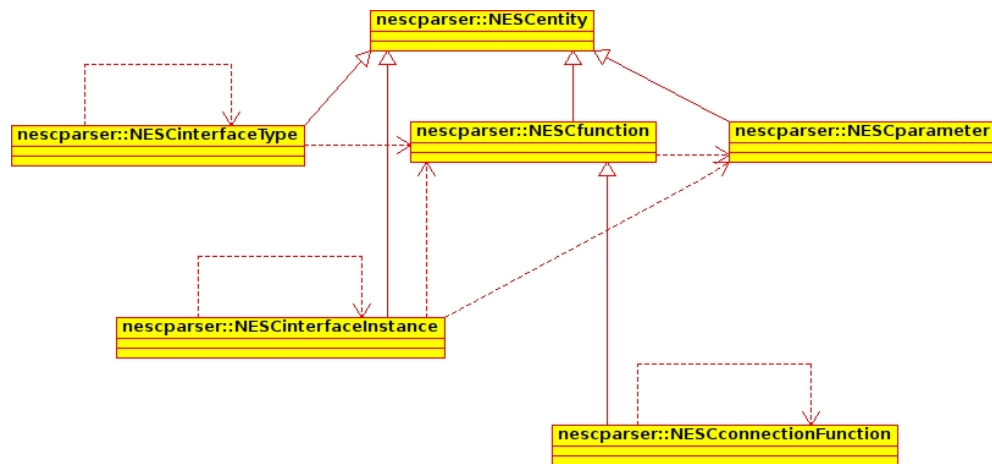


Figure 6.15: The NESCentity class hierarchy

compiler in a non TinyOS environment.

The applications contained in the two sets are listed below with a brief description of each application contained in the B set. For the A set the description is not reported, because it can be found in the directory tree of every TinyOS installation.

Set A:

- Blink
- BlinkTask
- CntToLeds
- CntToLedsAndRfm
- CntToRfm
- GenericBase
- GlowLeds
- GlowRadio
- Oscilloscope
- OscilloscopeRF

- RfmToLeds
- SecureTOSBase
- Sense
- SenseTask
- SenseToLeds
- SenseToRfm
- TestTinySec

Set B:

- AttributeTest  
This application is used to test the compiler's behaviour on nesC and gcc attributes.
- ComponentTest  
This application is used to see how the compiler works if the *components* nesC keyword is used.
- ConnectionTest  
This application is used to see the comportment of the compiler when connections are present in the sources.
- Default\_test  
This application is used to check the compiler's demeanour, if default nesC commands or events are present in the source code and only one component uses them.
- Default\_test\_switch  
This application is used to check the compiler's behaviour, if default nesC commands or events are present and different components use different implementations of those commands or events.
- EqualOperator\_first\_case  
This application is used to check the compiler's comportment, if the equal operator is used in a connection that connects an internal component with an external one.

- EqualOperator\_first\_case\_complicated

This application is used to check the compiler's comportment, if the equal operator is used in a connection that connects an internal component with an external one and the components are renamed by the *as nesC* keyword.

- EqualOperator\_second\_case

This application is used to check the compiler's comportment, if the equal operator is used in a connection that connects two external components.

- EqualOperator\_second\_case\_complicated

This application is used to check the compiler's comportment, if the equal operator is used in a connection that connects two external components and the components are renamed by the *as nesC* keyword.

- FanOut

This application is used to check the compiler's behaviour, if a component is connected to more than one other component.

- Global\_variable\_function\_arg

This application is used to check the compiler's demeanour, if a nesC file contains global variable or function declarations. In this case the resulting C file has to prefix each global function or variable name with the file name.

- Parameterized\_used\_interfaces

This application is used to check the compiler's comportment, if a nesC component uses parameterized interfaces.

- Tasks

This application is used to check the compiler's behaviour, if a nesC component uses tasks.

- Cfunctions

This application is used to check the compiler's demeanour, if a nesC file contains non global pure C functions or variables that are neither commands nor events.

- `First_parameters`

This application is used to check the compiler's comportment, if nesC commands or events contain parameters of fundamental types.

- `Complex_parameters`

This application is used to check the compiler's behaviour, if nesC commands or events contain parameters of directly derived types like for example arrays or pointers.

- `Implicit_with_equal_first_case`

This application is used to check the compiler's behaviour, if nesC components are involved in implicit connections, where the equal operator is used in a connection that connects an internal component with an external one.

- `Implicit_with_equal_second_case`

This application is used to check the compiler's behaviour, if nesC components are involved in implicit connections, where the equal operator is used in a connection that connects two internal components.





# Chapter 7

## Installation and Using

In this chapter a description of how to install the nesC-Erika software is given and a sample usage will be shown to test, if the environment has been installed correctly. First the required software is listed. After that, the needed installation steps are explained. Finally a sample application is compiled.

### 7.1 Software Requirements

Theoretically it is possible to install the nesC-Erika software both on a UNIX and Windows environment, but here only the installation on Windows XP will be described, because at the time of writing this work, only the *bat* version of the *rtdruid\_launcher* file was available. The *rtdruid\_launcher.bat* file is responsible of compiling the OIL file to a C file and to generate a makefile that can be executed after that.

For that reason the software requirements are the following.

- Windows XP professional
- Java JRE 1.6 or higher
- Cygwin <sup>1</sup>
- Erika Enterprise for AVR
- TinyOS 1.15
- TinyOS 2.0.1

---

<sup>1</sup>must be under C:\cygwin, very important!! In later versions this may change

- nesC 1.2
- nesC-Erika

## 7.2 Installation

This section supposes that the installation steps required to install Windows, the Java JRE and Cygwin are well known or at least that they can be read on other sources. For installing the Erika enterprise operating system it will be referred to the Evidence homepage [EVI], while for the installation of TinyOS 1.x, TinyOS 2.x and nesC 1.2 the following homepage is useful (<http://www.tinyos.net/tinyos-2.x/doc/html/upgrade-tinyos.html#tinyos2>). It describes very well how to have both a TinyOS 1.x directory tree in conjunction with a TinyOS 2.x tree. An important thing to note is that, as written in section 7.1, the cygwin environment must be installed in C:\cygwin otherwise the things don't work. Maybe this will change in further versions.

In the next section detailed instructions for the installation of nesc-Erika are given.

### 7.2.1 Installation of nesC-Erika

To install nesc-Erika the first thing to do is to download either the zip or tar.gz package available on the homepage. Next, the archive should be extracted in a directory. After that the software should be ready to be used. In the following a step by step guide of a method to install the nesC-Erika tar.gz package under Windows using the Cygwin environment is given. It is supposed that the package is called *nesc\_erika-0\_01-beta.tar.gz*.

1. The package *nesc\_erika-0\_01-beta.tar.gz* is downloaded in a directory. In the following it is supposed that this directory is called */cygdrive/g/home/finrod/workspace/backups/release*.
2. A Cygwin shell should be opened like in screenshot 7.1.
3. The */cygdrive/g/home/finrod/workspace/backups/release* directory, where the package was downloaded, should be entered as shown in screenshot 7.2.
4. The command  

```
tar xzf nesc_erika-0_01-beta.tar.gz
```

should be used to extract the archive as shown in screenshot 7.3.

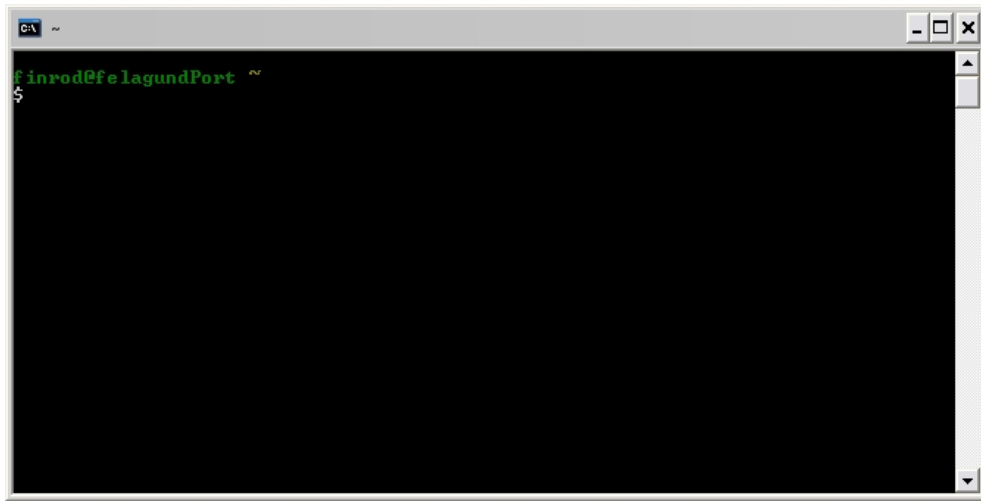


Figure 7.1: Cygwin shell

5. If the package was correctly installed, a subdirectory called *nesc\_erika* should have been created and the directory structure of this directory should look like in screenshot 7.4. In section 7.3 this directory is called the *nesC-Erika installation directory*.

In the following section a sample application is compiled to show how the translator is working.

## 7.3 Using - Sample compilation on an AVR target

To test if the environment has been installed correctly a sample application contained in the *nesc\_erika/Demo* directory can be compiled. It is supposed that the target board is Atmel's [ATM] STK500 in conjunction with Atmel's [ATM] STK501 board and the Atmega 128L microcontroller on it. In the following part of the section it is supposed to compile the BlinkTask application situated in the *nesc\_erika/Demo/BlinkTask* directory. To do so it is simply necessary to follow the successive steps. It is supposed that the steps shown in section 7.2.1 have been followed before.

1. The Makefile in the *nesc\_erika/Demo/BlinkTask* directory, should be opened in an editor as shown in screenshot 7.5

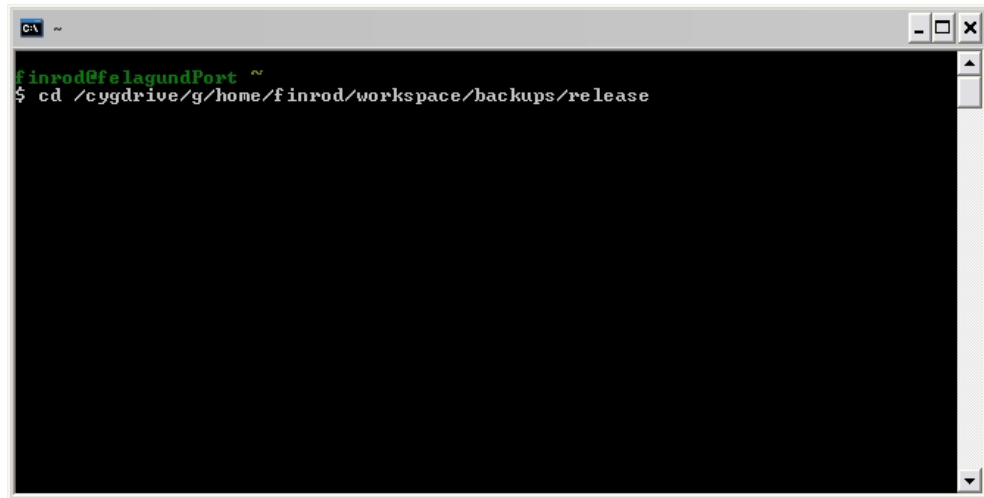


Figure 7.2: Entering the directory of the downloaded package

2. The RTDRUID variable should be edited to contain the exact absolute path to the *rtdruid\_launcher.bat* file. Normally the file is located in the *bin* subdirectory of the Erika installation directory. In sample screenshot 7.5 the absolute path to the *rtdruid\_launcher.bat* file is */cygdrive/c/Programmi/EvidenceAVR/bin/rtdruid\_launcher.bat*.
3. After having correctly edited the Makefile the nesC-Erika installation directory should be entered again as shown in screenshot 7.6.
4. The compilation command should be entered. To do so, the command `./ncc <top_configuration_file_name>`<sup>2</sup>

is used. The *top\_configuration\_file\_name* consists in the “main” nesC file name and normally it corresponds to the name of the directory, where the configuration files are contained with *nc* extension.

In the current example the command

`./ncc BlinkTask`

should be executed to compile the application as shown in screenshot 7.8.

---

<sup>2</sup>Actually, this works only with the applications contained in the Demo directory. For now this is only a prototype of the compiler, so this is done to show a faster demonstration. To change this behaviour the \$APP\_DIR variable contained in the ncc Perl script shown in screenshot 7.7 and contained in the nesC Erika installation directory can be edited.

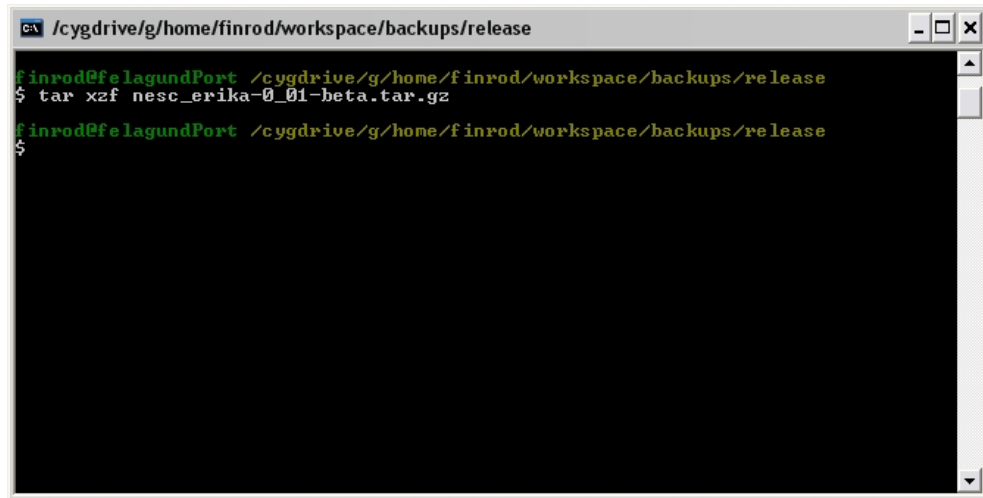
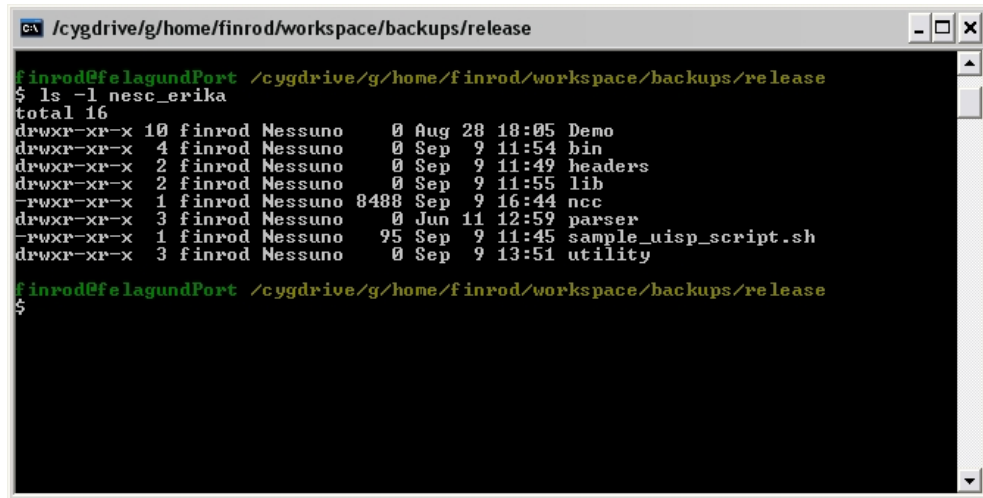


Figure 7.3: Extracting the archive

5. If everything compiled fine the message shown in screenshot 7.9 should be seen.
6. The important output files generated by the translator can be found in the *Demo/BlinkTask/* directory as shown in screenshot 7.10 and are summarized below.
  - *BlinkTask.nc.c*  
contains the main and other function definitions.
  - *handler.c*  
contains the interrupt handler routines.
  - *function\_declarations.h*  
contains the function declarations of all functions present in the application.
  - *preproc\_container.h*  
contains commands for the C preprocessor.
  - *threads.h*  
contains the declarations of the task used in the application.
  - *conf.oil*  
contains the *OIL* configuration of the application.

The other files that are generated can be ignored and in future versions they will not be visible anymore.



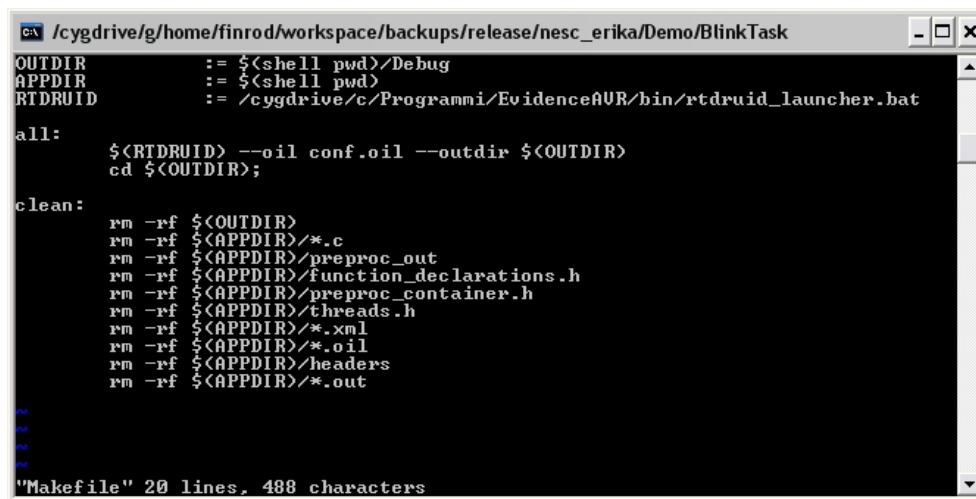
```

/cygdrive/g/home/finrod/workspace/backups/release
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release
$ ls -l nesc_erika
total 16
drwxr-xr-x 10 finrod Nessuno  0 Aug 28 18:05 Demo
drwxr-xr-x  4 finrod Nessuno  0 Sep  9 11:54 bin
drwxr-xr-x  2 finrod Nessuno  0 Sep  9 11:49 headers
drwxr-xr-x  2 finrod Nessuno  0 Sep  9 11:55 lib
-rwxr-xr-x  1 finrod Nessuno 8488 Sep  9 16:44 ncc
drwxr-xr-x  3 finrod Nessuno  0 Jun 11 12:59 parser
-rwxr-xr-x  1 finrod Nessuno 95 Sep  9 11:45 sample_uisp_script.sh
drwxr-xr-x  3 finrod Nessuno  0 Sep  9 13:51 utility
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release
$

```

Figure 7.4: The nesC-Erika installation directory tree

7. The *nesc\_erika/Demo/BlinkTask/Debug* directory contains the file generated by the *rtdruid\_launcher.bat* script. In particular a file called *avr.hex* can be found there as shown in screenshot 7.11. This is the final executable file of the application. This file is only suitable to be installed on the Atmega 128L microcontroller.
8. This file can be copied on the target board using the user's favorite programmer. The *sample\_uisp\_script.sh* is a script that contains an example command to copy the *avr.hex* file on the target board using the UISP [UIS] software.
9. An example usage of the *sample\_uisp\_script.sh* script is shown in screenshot 7.12. It takes an argument that consists in the path to the *avr.hex* file. In this case it is *Demo/BlinkTask/Debug/avr.hex*.
10. If desired, the *Demo/BlinkTask* directory can be entered again to delete all generated files using the  
*make clean*  
command.



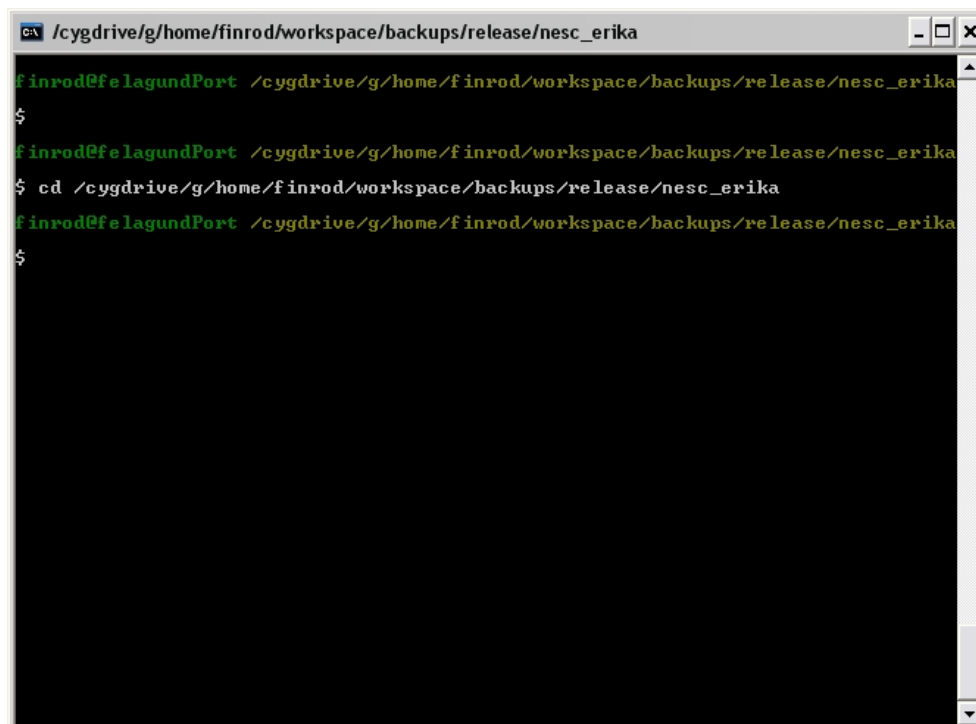
```
cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTask
OUTDIR      := $(shell pwd)/Debug
APPDIR      := $(shell pwd)
RTDRUID     := /cygdrive/c/Programmi/EvidenceAVR/bin/rtdruid_launcher.bat

all:
    $(RTDRUID) --oil conf.oil --outdir $(OUTDIR)
    cd $(OUTDIR);

clean:
    rm -rf $(OUTDIR)
    rm -rf $(APPDIR)/*.c
    rm -rf $(APPDIR)/preproc_out
    rm -rf $(APPDIR)/function_declarations.h
    rm -rf $(APPDIR)/preproc_container.h
    rm -rf $(APPDIR)/threads.h
    rm -rf $(APPDIR)/*.xml
    rm -rf $(APPDIR)/*.oil
    rm -rf $(APPDIR)/headers
    rm -rf $(APPDIR)/*.out

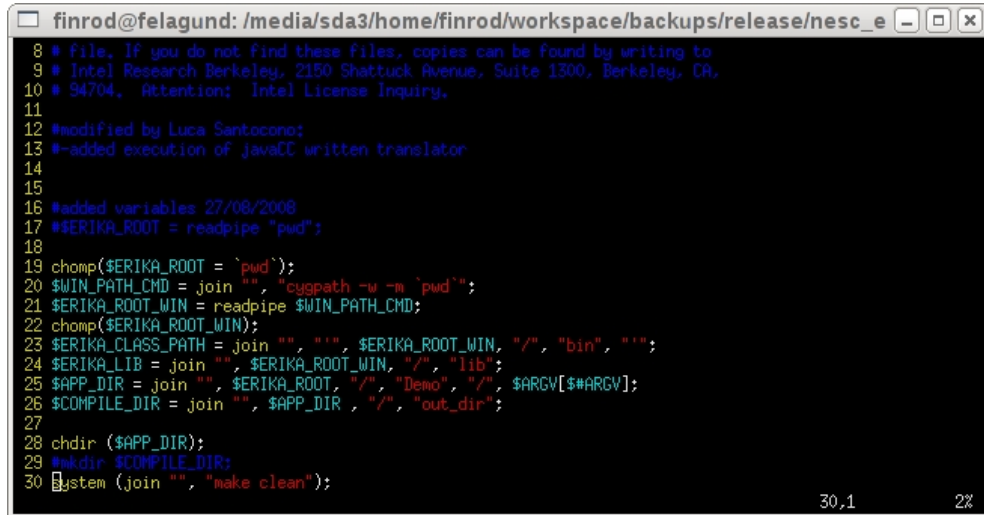
"Makefile" 20 lines, 488 characters
```

Figure 7.5: The application Makefile



```
cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$ cd /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$
```

Figure 7.6: Entering the nesC-Erika installation directory

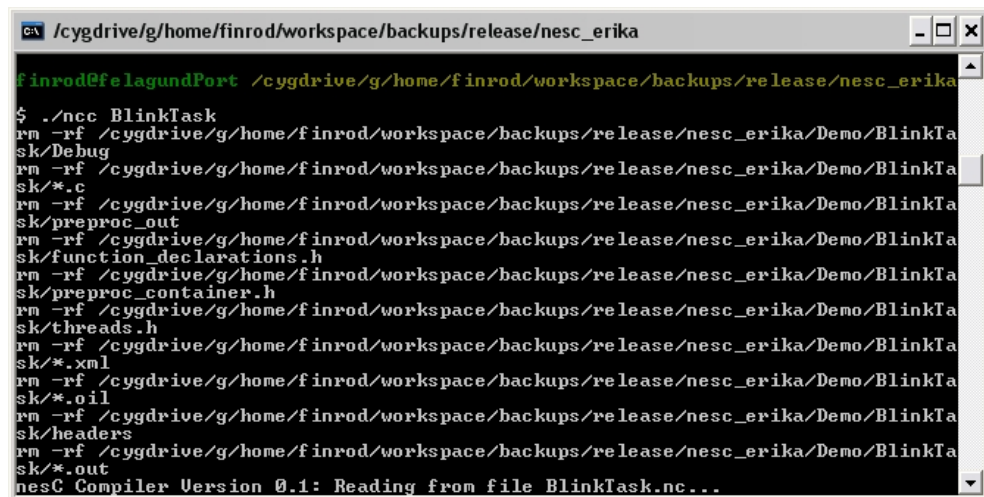


```

8 # file. If you do not find these files, copies can be found by writing to
9 # Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
10 # 94704. Attention: Intel License Inquiry.
11
12 #modified by Luca Santocoro;
13 #added execution of javaCC written translator
14
15
16 #added variables 27/08/2008
17 #ERIKA_ROOT = readpipe "pwd";
18
19 chomp($ERIKA_ROOT = `pwd`);
20 $WIN_PATH_CMD = join "", "cygpath -w -m `pwd`";
21 $ERIKA_ROOT_WIN = readpipe $WIN_PATH_CMD;
22 chomp($ERIKA_ROOT_WIN);
23 $ERIKA_CLASS_PATH = join "", "", $ERIKA_ROOT_WIN, "/", "bin", "";
24 $ERIKA_LIB = join "", $ERIKA_ROOT_WIN, "/", "lib";
25 $APP_DIR = join "", $ERIKA_ROOT, "/", "Demo", "/", $ARGV[$#ARGV];
26 $COMPILE_DIR = join "", $APP_DIR, "/", "out_dir";
27
28 chdir ($APP_DIR);
29 mkdir $COMPILE_DIR;
30 system (join "", "make clean");

```

Figure 7.7: The ncc Perl script



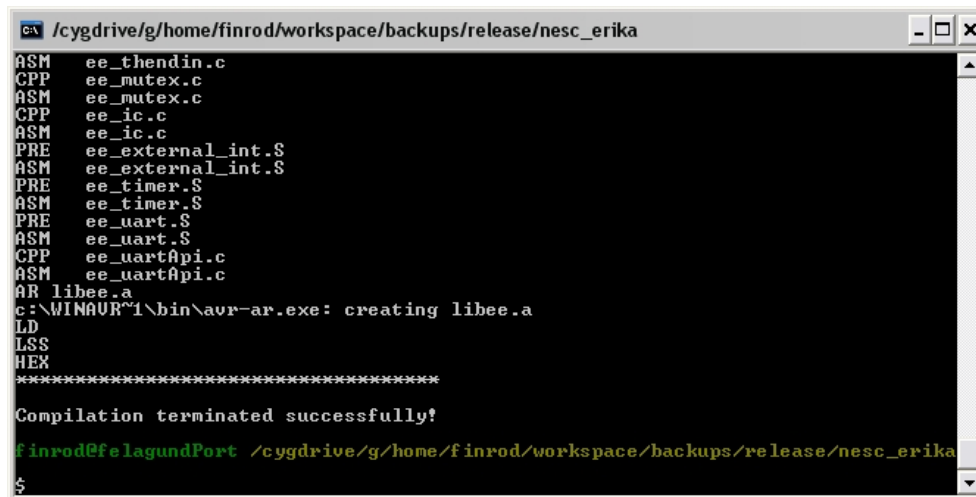
```

/cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$ ./ncc BlinkTask
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/Debug
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/*.c
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/preproc_out
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/function_declarations.h
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/preproc_container.h
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/threads.h
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/*.xml
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/*.oil
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/headers
rm -rf /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika/Demo/BlinkTa
sk/*.out
nesC Compiler Version 0.1: Reading from file BlinkTask.nc...

```

Figure 7.8: Compiling the sample application



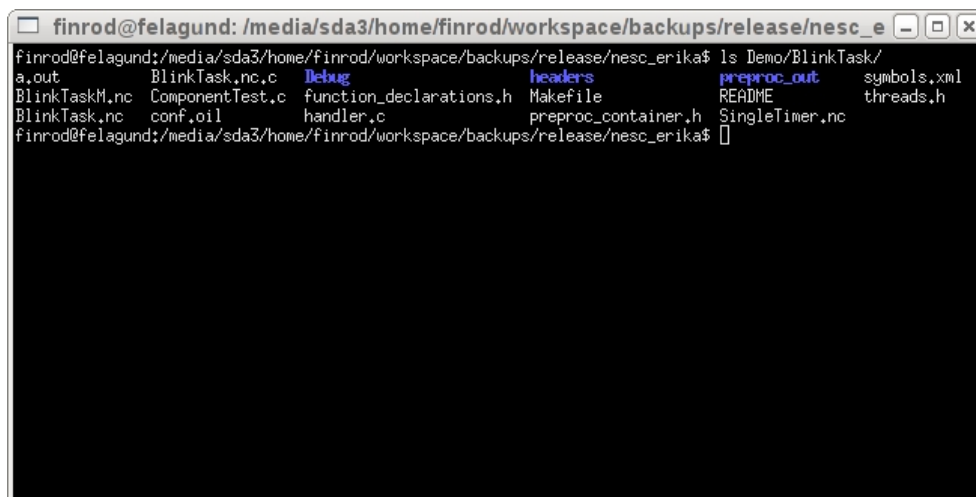


```

cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
ASM ee_thendin.c
CPP ee_mutex.c
ASM ee_mutex.c
CPP ee_ic.c
ASM ee_ic.c
PRE ee_external_int.S
ASM ee_external_int.S
PRE ee_timer.S
ASM ee_timer.S
PRE ee_uart.S
ASM ee_uart.S
CPP ee_uartApi.c
ASM ee_uartApi.c
AR libee.a
c:\WINAUR\1\bin\avr-ar.exe: creating libee.a
LD
LSS
HEX
*****
Compilation terminated successfully!
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$

```

Figure 7.9: Message after successful compilation

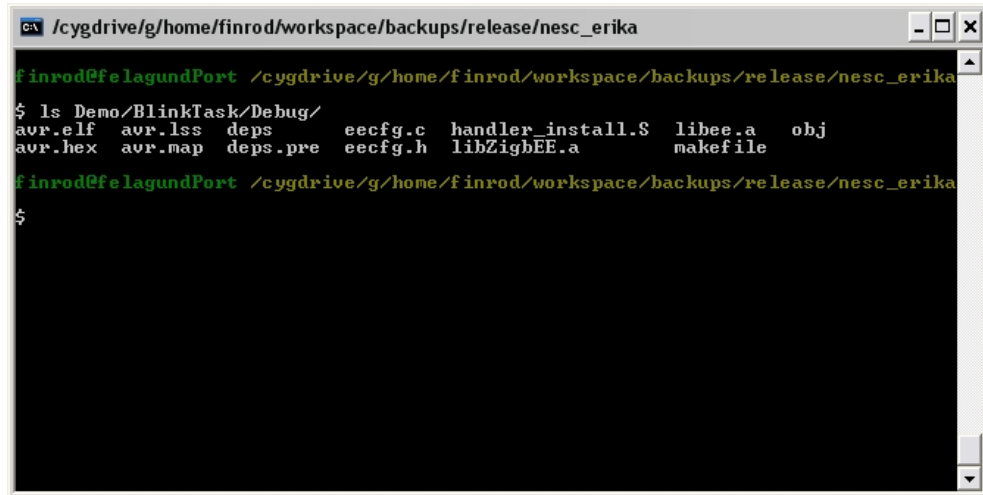


```

finrod@felagund: /media/sda3/home/finrod/workspace/backups/release/nesc_e
finrod@felagund:/media/sda3/home/finrod/workspace/backups/release/nesc_erika$ ls Demo/BlinkTask/
a.out      BlinkTask.nc.c  Debug          headers         preproc_out     symbols.xml
BlinkTaskM.nc  ComponentTest.c  function_declarations.h  Makefile        README          threads.h
BlinkTask.nc  conf.oil        handler.c       preproc_container.h  SingleTimer.nc
finrod@felagund:/media/sda3/home/finrod/workspace/backups/release/nesc_erika$

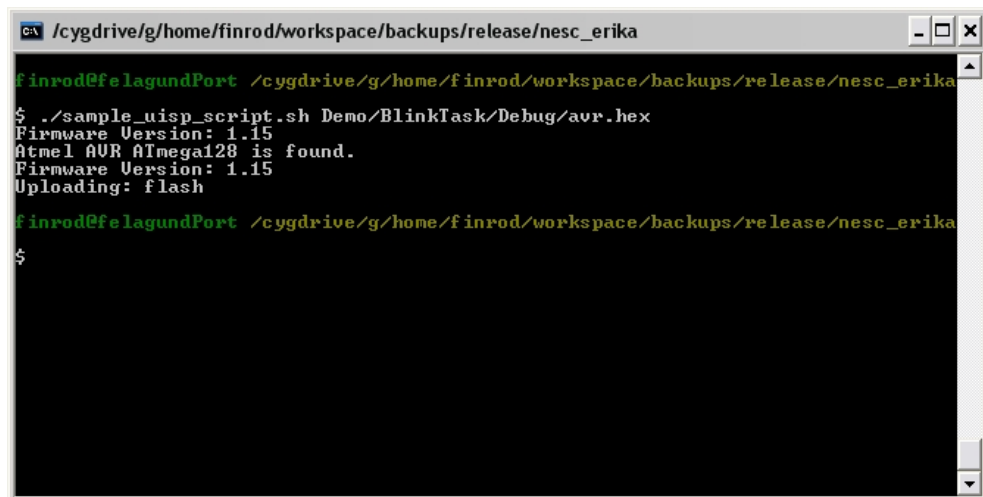
```

Figure 7.10: Files generated by the translator



```
cyg /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$ ls Demo/BlinkTask/Debug/
avr.elf  avr.lss  deps      eecfg.c  handler_install.$  libee.a  obj
avr.hex  avr.map  deps.pre  eecfg.h  libZigbEE.a        makefile
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$
```

Figure 7.11: The *avr.hex* file located in the *nesc\_erika/Demo/BlinkTask/Debug* directory.



```
cyg /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$ ./sample_uisp_script.sh Demo/BlinkTask/Debug/avr.hex
Firmware Version: 1.15
Atmel AVR ATmega128 is found.
Firmware Version: 1.15
Uploading: flash
finrod@felagundPort /cygdrive/g/home/finrod/workspace/backups/release/nesc_erika
$
```

Figure 7.12: Using the script *sample\_uisp\_script.sh*

# Chapter 8

## Experiments

In this chapter experiments on a real hardware platform are reported. The first section gives an overview of the hardware used for the test, like the development board and microcontroller used. The second section will describe the testing applications chosen, the third section describes the selected In-System Programmer, the forth the used cross compiler, whereas the fifth and final section describes the results obtained. In the results there will be also a comparison between the files generated by the nesC-TinyOS compiler and the files generated by the nesC-Erika compiler that are needed to obtain the final executable in terms of number of code lines and size. The final executables itself will be compared as well in this way.

### 8.1 Target Hardware

To test the software, Atmel's ATmega128 AVR [ATM] microcontroller on the STK500 and STK501 development board, has been chosen as target hardware platform.

#### 8.1.1 STK500 development board

The Atmel AVR STK500 is a starter kit and development system for Atmel's AVR Flash microcontrollers. The system comprises of a single board module which features IC <sup>1</sup> sockets for all the popular dual-in-line AVR devices. It makes it possible to in-system program (ISP) AVR devices on the STK500 and also to use the board to ISP a device on a separate Target System. The STK500 can be used as an evaluation board by connecting up the various

---

<sup>1</sup>Integrated circuit

on-board peripherals including LED's, push buttons and Serial Dataflash. The features are summarized in the following list.



Figure 8.1: The STK500 board

- Dual-in-line (DIL) Sockets to accommodate 8, 20, 28 and 40 pin AVR microcontrollers
- Supports Serial In-System Programming (ISP) of AVR microcontrollers on the STK500 Board
- Supports In-System Programming of AVR microcontrollers in an External Target System
- Supports Parallel and Serial High-voltage programming of AVR microcontrollers on the STK500 Board
- 8 x Push buttons
- 8 x LED's for general use
- All AVR I/O ports accessible through pin header connectors
- Expansion connectors for plug-in modules (e.g. STK501)
- Prototyping area

- On-board 2 Megabit Serial Dataflash for non-volatile data storage
- Flexible clocking, voltage and reset system
- RS-232 Interface to PC for Programming and Configuration
- Second RS-232 port for user application
- External power supply required (9V - 12V DC)
- AT89(L)S - 8051, AT90S AVR, ATmega AVR, ATtiny AVR FLASH microcontroller Family support

### 8.1.2 STK501 development board

The STK501 board is a top module designed to add ATmega103, ATmega64 and ATmega128 support to the STK500 development board from Atmel Corporation [ATM]. With this board the STK500 is extended to support all current AVR devices in a single development environment. In addition to adding support for new devices, it also adds new support for peripherals previously not supported by the STK500. An additional RS-232 port and External SRAM interface is among the new features. Devices with Dual UART or XRAM interface can all take advantage of the new resources on the STK501 board.

In the following list the features of this board are summarized.

- STK500 Compatible
- AVR Studio Compatible
- Supports ATmega103, ATmega64 and ATmega128
- Zero Insertion Force Socket for TQFP packages
- TQFP Footprint for Emulator Adapters
- Supports all added features in ATmega128
- JTAG connector for On-Chip Emulation using JTAG ICE (ATmega128)
- Additional RS-232C port
- Adds XRAM support to the STK500 board. (usable for all devices with XRAM interface)
- On board 32kHz clock oscillator for easy RTC implementations

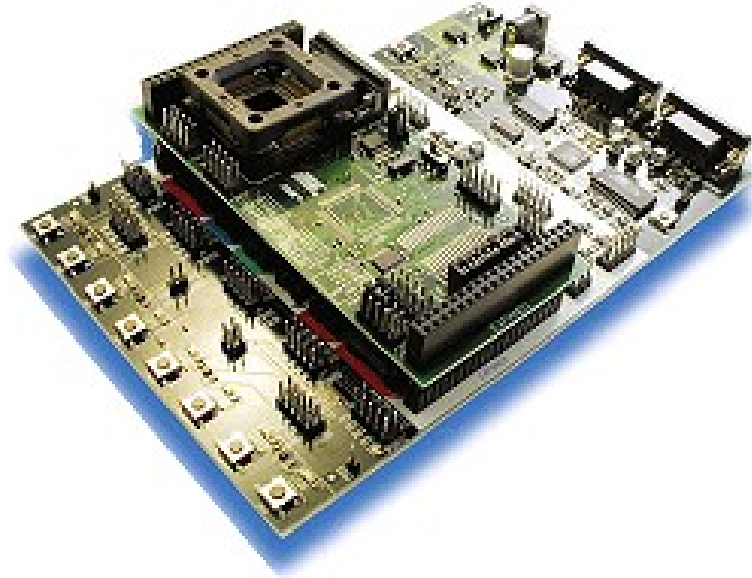


Figure 8.2: The STK501 board on top of the STK500

### 8.1.3 ATmega128 AVR microcontroller

In general the AVR family of Atmel's microcontroller is a modified Harvard architecture machine with program and data stored in separate physical memory systems that appear in different address spaces but having the ability to read data items from program memory using special instructions. The ATmega128 AVR in particular, is a microcontroller able to run at 16MHz.

The main characteristics of this microcontroller are summed up below.

- 128-Kbyte self-programming Flash Program Memory
- 4-Kbyte SRAM
- 4-Kbyte EEPROM
- 8 Channel 10-bit A/D-converter
- JTAG interface for on-chip-debug
- Up to 16 MIPS throughput at 16 MHz
- 2.7 - 5.5 Volt operation.

## 8.2 In-System Programmer

The selected in-system programmer to copy the final executable on the target platform is the UISP - AVR In-system Programmer [UIS]<sup>2</sup>. It can be run on UNIX systems as well as on Windows platforms using the cygwin environment. UISP allows to program the microcontroller through the parallel or serial port of the computer. It is possible to use the many wirings available. The most important options are listed below.

- `-dprog=avr910|pavr|stk500`

avr910

Standard Atmel Serial Programmer/Atmel Low Cost Programmer

pavr

<http://www.avr1.org/pavr/pavr.html>

stk500

Atmel STK500

- `-dpart=part`

This option sets the target abbreviated name or number. For some programmers, if `-dpart` is not given, the list of programmer's supported devices is shown. For auto-select `-dpart=auto` can be given, but it does not work with all programmers, so it is recommended to always specify a target device explicitly.

- `-dserial=device name`

In a Unix environment the serial interface as `/dev/ttyS*` can be set (default `/dev/avr`). If the software is run on Windows under cygwin than the interface is set as `/dev/com*`.

- `-upload`

The "input\_file" specified after the *if* option is uploaded to the AVR memory.

- `if=filename`

The input file for the `-upload` and `-verify` functions in Motorola S-records (S1 or S2) or 16 bit Intel format.

- `-erase`

Erase device

---

<sup>2</sup>acronym that stands for "micro In System Programmer", the u symbolizes a greek  $\mu$

- `-verify`

The "input\_file" processed after the `-upload` option is verified.

### 8.3 Cross compiler

The selected cross compiler is the WinAVR [WAV] compiler, because it is the most suitable to be run from a Windows environment. It consists in a suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors. It includes the GNU GCC compiler for C and C++.

### 8.4 nesC-TinyOS vs nesC-Erika

In this section a comparison between the nesC-TinyOS and nesC-Erika compiler is shown. The tests are done on demo applications available in the *Demo* directory of the nesC-Erika installation directory <sup>3</sup>. This applications are also available in the TinyOS directory. For each tested application four tables are reported.

The first one counts the code lines and the memory size of the files generated by nesC-Erika. NesC-Erika generates more than one intermediate C file and therefore the code lines and memory size are counted separately for each single file. In the last row of the tabel the memory usage of the final executable is shown.

The second table shows the memory usage and number of code lines of the files generated by nesC-TinyOS. NesC-TinyOS generates only one intermediate C file and it is normally called *<application\_name>.c*. In the case of Blink for example this file is called *Blink.c*. In the last row the memory usage of the final executable generated by nesC-TinyOS is shown.

The third table shows a comparison between the intermediate files generated by nesC-TinyOS and nesC-Erika. The memory usage and number of code lines of the nesC-TinyOS intermediate file are compared against the memory usage and number of code lines of all intermediate files generated by nesC-Erika together. This time the columns represent the files and the rows represent the resources used, in terms of memory usage and number of code lines. In this table there are also two final columns showing respectively the absolute and relative <sup>4</sup> differences.

<sup>3</sup>see section 1.4 for the meaning of nesC-Erika installation directory.

<sup>4</sup>The relative difference is calculated with the formula

$d_r = |x - y| / \max(|x|, |y|)$  and expressed in percentage.



The forth table compares the memory usage of the executable generated by nesC-TinyOS against the one generated by nesC-Erika.

### Blink

It is a basic application that starts a 1Hz timer and toggles the red LED every time it fires.

The four tables (8.1, 8.2, 8.3 and 8.4) compare the different results obtained by using the nesC-TinyOS and nesC-Erika compiler for Blink.

	Memory usage	Code lines
Blink.nc.c	29 KiloByte	1621
function_declarations.h	6 KiloByte	133
preproc_container.h	40 Byte	2
handler.c	95 Byte	5
threads.h	100 Byte	3
conf.oil	1,8 KiloByte	78
nesC-Erika executable	11 KiloByte	-

Table 8.1: Files generated by nesC-Erika for Blink

	Memory usage	Code lines
nesC-TinyOS intermediate file	44 KiloByte	2006
nesC-TinyOS executable	14 KiloByte	-

Table 8.2: Files generated by nesC-TinyOS for Blink

### BlinkTask

BlinkTask is a basic application that toggles the leds on the mote on every clock interrupt. The difference between Blink and BlinkTask is how the Clock.fire() event is handled. BlinkTask offloads processing to a task which controls the LEDs. Blink controls the LEDs directly in the event handler thereby not returning from the event until the LEDs have been toggled. The clock interrupt is scheduled to occur every second. The initialization of the clock can be seen in the Blink initialization function, StdControl.start().

	nesC-TinyOS intermediate file	nesC-Erika intermediate files	absolute difference	relative difference
Memory usage	44 KiloByte	37,035 KiloByte	6,965 KiloByte	15,83%
Code lines	2006	1842	164	8,18%

Table 8.3: nesC-TinyOS vs. nesC-Erika intermediate files for Blink

	nesC-TinyOS executable	nesC-Erika executable	absolute difference	relative difference
Memory usage	14 KiloByte	11 KiloByte	3 KiloByte	21,43%

Table 8.4: nesC-TinyOS vs. nesC-Erika executables for Blink

The four tables (8.5, 8.6, 8.7 and 8.8) compare the different results obtained by using the nesC-TinyOS and nesC-Erika compiler for BlinkTask.

	Memory usage	Code lines
BlinkTask.nc.c	29 KiloByte	1633
function_declarations.h	6 KiloByte	133
preproc_container.h	40 Byte	2
handler.c	99 Byte	5
threads.h	131 Byte	4
conf.oil	1,9 KiloByte	85
nesC-Erika executable	12 KiloByte	-

Table 8.5: Files generated by nesC-Erika for BlinkTask

### CntToLeds

CntToLeds maintains a counter on a 4Hz timer and displays the lowest three bits of the counter value. The red LED is the least significant of the bits, while the yellow is the most significant. The four tables (8.9, 8.10, 8.11 and 8.12) compare the different results obtained by using the nesC-TinyOS and nesC-Erika compiler for CntToLeds.

	Memory usage	Code lines
nesC-TinyOS intermediate file	44 KiloByte	2013
nesC-TinyOS executable	14 KiloByte	-

Table 8.6: Files generated by nesC-TinyOS for BlinkTask

	nesC-TinyOS intermediate file	nesC-Erika intermediate files	absolute difference	relative difference
Memory usage	44 KiloByte	37.17 KiloByte	6,83 KiloByte	15,52%
Code lines	2013	1862	151	7,50%

Table 8.7: nesC-TinyOS vs. nesC-Erika intermediate files for BlinkTask

### GlowLeds

This application increments or decrements the Leds intensity based on the value of a state variable every time the timer fires.

The four tables (8.13, 8.14, 8.15 and 8.16) compare the different results obtained by using the nesC-TinyOS and nesC-Erika compiler for GlowLeds.

	nesC-TinyOS executable	nesC-Erika executable	absolute difference	relative difference
Memory usage	14 KiloByte	12 KiloByte	2 KiloByte	14,29%

Table 8.8: nesC-TinyOS vs. nesC-Erika executables for BlinkTask

	Memory usage	Code lines
CntToLeds.nc.c	30 KiloByte	1701
function_declarations.h	6.4 KiloByte	140
preproc_container.h	40 Byte	2
handler.c	96 Byte	5
threads.h	131 Byte	4
conf.oil	1.9 KiloByte	85
nesC-Erika executable	12 KiloByte	-

Table 8.9: Files generated by nesC-Erika for CntToLeds

	Memory usage	Code lines
nesC-TinyOS intermediate file	50 KiloByte	2337
nesC-TinyOS executable	16 KiloByte	-

Table 8.10: Files generated by nesC-TinyOS for CntToLeds

	nesC-TinyOS intermediate file	nesC-Erika intermediate files	absolute difference	relative difference
Memory usage	50 KiloByte	38.567 KiloByte	11,433 KiloByte	22,87%
Code lines	2337	1937	400	17,12%

Table 8.11: nesC-TinyOS vs. nesC-Erika intermediate files for CntToLeds

	nesC-TinyOS executable	nesC-Erika executable	absolute difference	relative difference
Memory usage	16 KiloByte	12 KiloByte	4 KiloByte	25%

Table 8.12: nesC-TinyOS vs. nesC-Erika executables for CntToLeds

	Memory usage	Code lines
GlowLeds.nc.c	33 KiloByte	1767
function_declarations.h	6,4 KiloByte	139
preproc_container.h	40 Byte	2
handler.c	98 Byte	5
threads.h	132 Byte	4
conf.oil	1,9 KiloByte	85
nesC-Erika executable	13 KiloByte	-

Table 8.13: Files generated by nesC-Erika for GlowLeds

	Memory usage	Code lines
nesC-TinyOS intermediate file	50 KiloByte	2162
nesC-TinyOS executable	16 KiloByte	-

Table 8.14: Files generated by nesC-TinyOS for GlowLeds

	nesC-TinyOS intermediate file	nesC-Erika intermediate files	absolute difference	relative difference
Memory usage	50 KiloByte	41.57 KiloByte	8,43 KiloByte	16,86%
Code lines	2162	2002	160	7,4%

Table 8.15: nesC-TinyOS vs. nesC-Erika intermediate files for GlowLeds

	nesC-TinyOS executable	nesC-Erika executable	absolute difference	relative difference
Memory usage	16 KiloByte	13 KiloByte	3 KiloByte	18,75%

Table 8.16: nesC-TinyOS vs. nesC-Erika executables for GlowLeds

# Chapter 9

## Future work

In the present chapter some suggestions on which kind of work is still needed on the compiler are given.

### 9.1 Erika nesC library

As said in other chapters a nesC-Erika library would be needed to obtain a ideally working nesC-TinyOS to nesC-Erika translator. One of the main objectives of the thesis was to construct a tool that can automatically translate a nesC-TinyOS application to a nesC-Erika one. This is without such a library difficult to realize. As described earlier this library can be substituted by a set of C header files. In any case a set of Erika system calls that are equivalent to TinyOS system calls are needed, otherwise every single application that is going to be translated can contain a system call that is not substituted and so the whole application is not independent from TinyOS.

### 9.2 Going away from nesC-TinyOS

In the future it is important to try to build a translator that is able to work without the help of nesC-TinyOS. This means that an error checker capable of catching all kind of errors is needed. In addition a C preprocessor directly integrated in the compiler would be very useful for that purpose. This would allow to avoid having to parse the XML file for collecting all the types defined by typedef. Finally, the objects containing important cross component information involved in the code generation process should be generated not by parsing the XML file, but by parsing the source file and the parse tree.

### 9.3 TinyOS 2 support

For now the translator supports only applications that are written for the TinyOS 1.x operating system that uses the nesC 1.1.x language specification. Applications written for TinyOS 2.x, which uses the newer nesC 1.2.x language can be parsed, but no code is generated for those applications yet. Between nesC 1.1.x and 1.2.x there are many new language constructs available. Those new constructs should be supported by the code generators available for nesC-Erika to make it possible to compile TinyOS 2.x applications.

A summary of the most important changes between nesC 1.1.x and 1.2.x can be found in the following lines.

- Generic interfaces defined by the *generic interface* keywords.
- Generic components defined by the *generic module* or *generic configuration* keywords.
- Generic interfaces or generic components are instantiated by the *new* keyword.
- An interface definition can now have arguments.
- Binary components: programs can now use components defined in binary form.
- External types: types with a platform-independent representation and no alignment representation can now be defined in nesC.
- Attributes: declarations may be decorated with attributes. The use of *\_\_attribute\_\_* for nesC-specific features is deprecated <sup>1</sup>.
- The *includes* keyword is deprecated and components can be preceded by arbitrary C declarations and macros. Particularly, instead of *includes* the standard C *#include* keyword is used.
- The *return* keyword can be used within *atomic* statements. Hereby the atomic statement is automatically terminated by the *return* keyword.

---

<sup>1</sup>used only for gcc attributes anymore, instead constructs of the form *@identifier(initializer-list)* are used for nesC v1.2.



# Bibliography

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principle, Techniques and Tools*. Addison-Wesley, 1986.
- [ATM] <http://www.atmel.com/> - Atmel Corporation - Design and Manufacture of Advanced Semiconductors, Official Website.
- [CAD] <http://cadena.projects.cis.ksu.edu/> - Cadena homepage.
- [CYG] <http://www.cygwin.com/> - Cygwin Information and Installation, Official Website.
- [DRU08] *RT-Druid Code generator Plugin reference manual*, 2008.
- [ECL] <http://www.eclipse.org/> - Eclipse, Official Website.
- [ERI08] *ERIKA Enterprise Manual*, 2008.
- [EVI] <http://www.evidence.eu.com/> - Evidence SRL Official Website.
- [GLCB03] David Gay, Philip Levis, David Culler, and Eric Brewer. *nesC 1.1 Language Reference Manual*, 2003.
- [GLCB05] David Gay, Philip Levis, David Culler, and Eric Brewer. *nesC 1.2 Language Reference Manual*, 2005.
- [GNU] <http://gcc.gnu.org/> - GCC, the GNU Compiler Collection.
- [JCCa] <https://javacc.dev.java.net/> - JavaCC Official Website.
- [JCCb] [http://pagesperso-orange.fr/eclipse\\_javacc/](http://pagesperso-orange.fr/eclipse_javacc/) - JavaCC Eclipse Plug-in, Official Website.
- [KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

- [Lev06] Philip Levis. *TinyOS Programming*, 2006.
- [Man06] Marco Maniscalco. *Paradigms for using semantic actions and code execution with JavaCC*, 2006.
- [MES] <http://www.meshnetics.com/tinyos/> - MeshNetics/TinyOS.
- [NCU] <http://www.ipvs.uni-stuttgart.de/abteilungen/vs/abteilung/mitarbeiter/eigenes/lachenas/ncunit/start/#Usage> - nCUnit: A Unit Testing Framework for nesC.
- [NES] <http://sourceforge.net/projects/nesc> - nesC: A Programming Language for Deeply Networked Systems, Official Website.
- [Nor07] Theodore S. Norvell. *The JavaCC FAQ*, 2007.
- [OSE] <http://www.osek-vdx.org/> - OSEK VDX Portal, Official Website.
- [RCSa] <http://www.cs.purdue.edu/homes/trinkle/RCS/> - Official RCS Homepage.
- [RCSb] <http://www.madboa.com/geek/rcs/> - RCS HOWTO.
- [RET] <http://retis.sssup.it/> - Real-Time Systems Laboratory of the Scuola Superiore Sant'Anna di Pisa, Official Website.
- [SUN] <http://java.sun.com/> - Developer Resources for Java Technology, Official Website.
- [TDT] <http://sourceforge.net/projects/tinydt> - TinyDT Eclipse plugin.
- [TID] <http://tide.ucd.ie/> - TinyosIDE Eclipse plugin.
- [TOS] <http://www.tinyos.net/> - TinyOS Official Website.
- [UIS] <http://www.nongnu.org/uisp/> - UISP - AVR In-System Programmer, Official Website.
- [WAV] <http://winavr.sourceforge.net/> - AVR-GCC for Windows, Official Website.
- [YET] <http://tos-ide.ethz.ch/wiki/index.php> - TinyDT Eclipse plugin.