Porting TinyOS to an Amulet2e Sensor Mote

by

Chris Sienkiewicz

Supervisor: Nick Filer

2010
Abstract

This report details the work carried out for a third year project in the School of Computer Science, at the University of Manchester.

TinyOS is an industry standard operating system for wireless sensor motes: small computers designed to collect data and wirelessly communicate with one another. The Amulet2e is one such mote, and a successful port of TinyOS was produced for it. Support for non-standard mote functionality was also added, enabling the full use of the Amulet2e sensor mote hardware from TinyOS. Basic simulation support for the lower level systems of the mote was created, and an automated build chain was developed to ease the development process.

Whilst porting TinyOS, it was noted that it can be difficult to produce programs for TinyOS, especially for developers new to the operating system. A new language was designed and implemented to help alleviate this problem. This language was then used to create some demonstration applications, which allowed for some simple experiments to be carried out using TinyOS on the Amulet2e sensor mote.
Acknowledgements

I would like to thank Nick Filer for his invaluable help, support, and continuous enthusiasm throughout this project.

I would also like to thank my family and friends, for their continued support whilst writing this report.
# Contents

Abstract i  
Acknowledgements iii  
Contents v  
List of Symbols and Abbreviations ix  

## 1 Introduction

1.1 Introduction Overview ........................................... 1  
1.2 What are Sensor Motes? ........................................ 1  
1.3 What are TinyOS and Amulet2e? .............................. 2  
1.4 Project Overview and Rationale .............................. 2  
1.5 Aims and Objectives .......................................... 3  
1.6 Literature Review ............................................ 3  
1.6.1 TinyOS Extension Proposals ............................. 3  

## 2 TinyOS and nesC

2.1 Introduction ..................................................... 5  
2.2 TinyOS ......................................................... 5  
2.2.1 Introduction to TinyOS ................................... 5  
2.2.2 Programs .................................................. 5  
2.3 nesC ........................................................... 6  
2.3.1 Introduction to nesC ..................................... 6  
2.3.2 Components and Wirings ................................ 7  
2.3.3 Concurrency .............................................. 8  
2.3.4 Optimisation ............................................. 8  
2.3.5 Example Program ........................................ 9  
2.4 TinyOS Architecture .......................................... 11  
2.4.1 Platforms, Chipsets and Sensor Boards .................. 11  
2.4.2 Hardware Astraction Architecture ....................... 11  
2.5 Toolchain ..................................................... 12
CONTENTS

2.5.1 Build System ........................................... 12
2.5.2 nesC compiler chain ................................... 13

3 The Amulet2e Sensor Mote .......................... 15
  3.1 Introduction ........................................... 15
  3.2 Features ................................................. 16
  3.3 Existing Code .......................................... 16
  3.4 Toolchain ............................................... 17

4 Porting TinyOS ........................................... 19
  4.1 Introduction ........................................... 19
  4.2 Hardware Presentation Layer ....................... 21
    4.2.1 Component Layout .................................. 21
    4.2.2 Interrupt Handling ................................. 21
    4.2.3 Changing Power States ............................ 23
  4.3 Hardware Adaptation Layer ......................... 24
    4.3.1 Component Layout .................................. 24
    4.3.2 Radio Communication ............................... 24
    4.3.3 Timer .............................................. 26
    4.3.4 Serial Communication .............................. 27
    4.3.5 I2C Bus ............................................ 27
  4.4 Hardware Interface Layer ............................ 28
    4.4.1 Timer .............................................. 28
    4.4.2 LEDs ............................................... 28
    4.4.3 Radio Communication ............................... 29
  4.5 Toolchain ............................................... 29
    4.5.1 Basic Compilation .................................. 29
    4.5.2 Simulate ............................................ 30
    4.5.3 Download .......................................... 31
    4.5.4 Docs ............................................... 32
  4.6 Result .................................................. 32

5 Hench .................................................... 33
  5.1 Introduction ........................................... 33
  5.2 The need for a new language ........................ 33
  5.3 Design .................................................. 33
    5.3.1 Specifying the Interfaces Used .................... 34
    5.3.2 Event Handling ..................................... 35
    5.3.3 Other Notes ........................................ 35
    5.3.4 Example Program ................................... 35
  5.4 Implementation ......................................... 36
    5.4.1 The Hench Grammar .................................. 36
    5.4.2 The Compiler ........................................ 37
    5.4.3 HenchC Script ...................................... 40
5.5 Results ......................................................... 41

6 Demonstration Applications and Simple Experiments  43
   6.1 Introduction ............................................. 43
   6.2 Amulet2 Tester .......................................... 43
   6.3 Example Sensor Network .............................. 45
   6.4 Network Experiments ................................. 47
       6.4.1 Battery Life ....................................... 47
       6.4.2 Radio Errors ....................................... 47
       6.4.3 Error Rates at Different Distances .......... 48

7 Further Work  .................................................. 49
   7.1 Introduction ............................................. 49
   7.2 Improvements to Hench ............................... 49
   7.3 Serial Baud Selection ................................. 49
   7.4 Debugging Support .................................... 50
   7.5 Low Battery Warning ................................. 50

8 Conclusion ....................................................... 51
   8.1 Introduction ............................................. 51
   8.2 Aims and Objectives .................................. 51
   8.3 Challenges Faced ...................................... 52
   8.4 Project Evaluation .................................... 52

Bibliography ..................................................... 53
## List of Symbols and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
<td>page 37</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare Register</td>
<td>page 21</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
<td>page 43</td>
</tr>
<tr>
<td>EE</td>
<td>Electronically Erasable</td>
<td>page 16</td>
</tr>
<tr>
<td>FIFO</td>
<td>First in, First Out</td>
<td>page 8</td>
</tr>
<tr>
<td>GDB</td>
<td>Gnu Project Debugger</td>
<td>page 17</td>
</tr>
<tr>
<td>GCC</td>
<td>Gnu Compiler Collection</td>
<td>page 13</td>
</tr>
<tr>
<td>HAA</td>
<td>Hardware Abstraction Architecture</td>
<td>page 11</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Adaptation Layer</td>
<td>page 11</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware Interface Layer</td>
<td>page 11</td>
</tr>
<tr>
<td>HPL</td>
<td>Hardware Presentation Layer</td>
<td>page 11</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
<td>page 16</td>
</tr>
<tr>
<td>LDR</td>
<td>Light Dependant Resistor</td>
<td>page 16</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
<td>page 16</td>
</tr>
<tr>
<td>NCC</td>
<td>The nesC Compiler for TinyOS</td>
<td>page 13</td>
</tr>
<tr>
<td>NESCC</td>
<td>The nesC Compiler</td>
<td>page 13</td>
</tr>
<tr>
<td>RDI</td>
<td>Remote Debug Interface</td>
<td>page 17</td>
</tr>
<tr>
<td>TEP</td>
<td>TinyOS Extension Proposal</td>
<td>page 3</td>
</tr>
<tr>
<td>TIM</td>
<td>Timer Register</td>
<td>page 21</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver and Transmitter</td>
<td>page 16</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
<td>page 1</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction Overview

This chapter introduces the project and the accompanying report, as well as presenting some basic background information for readers unfamiliar with the topic area.

1.2 What are Sensor Motes?

A sensor mote, also known as a sensor node, is a small battery powered computer (an example is shown in figure 1.1). It has sensors attached for sampling the external environment and can communicate with other sensor motes nearby. Due to their size and power source, motes usually have a very constrained set of computational resources, including slow processing units and small amounts of memory.

![Figure 1.1: A Telos Mote, without battery pack][1]

Mote to mote communication is usually achieved with the use of a radio transceiver allowing for wireless communication. When several motes communicate wirelessly they form a Wireless Sensor Network (WSN).
WSNs can be distributed across a large area, allowing data to be captured about a whole area, rather than at a single point. As the motes are not static, the network is usually \textit{ad hoc} and self-organising. This allows the positions of the motes to be moved to arbitrary positions, as well as new motes added or existing motes removed, without the need to manually reconfigure the network.

Often a mote is connected to a computer, and acts as a base station. All data in the network is sent to this base mote, which then relays it to the computer for analysis (as in figure 1.2). Alternatively, when in the field, the mote may store the data for later retrieval and analysis.

1.3 What are TinyOS and Amulet2e?

TinyOS is an industry standard operating system, developed specifically for WSNs. The operating system and its associated language are discussed in Chapter 2.

The Amulet2e mote is a sensor mote developed in the School of Computer Science at the University of Manchester. It is discussed in detail in Chapter 3.

1.4 Project Overview and Rationale

The primary objective of the project was to port TinyOS to the Amulet2e sensor mote. This would allow researchers and students within the school to create and experiment upon example sensor networks based on the hardware already residing, but currently unused, within the school.

While multiple WSN operating systems exist, TinyOS was chosen due to its established position within the field of WSNs. This allows developers working with the system to take advantage of a large base of existing code and tools, as well as allowing the network programs created to run on a large range of other hardware. This phase of the project is described in Chapter 4.

The second initial objective was simply to extend the project as time allowed. During the porting phase of the project it was noted that development for TinyOS can be somewhat complex, especially
for developers new to TinyOS. It was thus decided as a secondary objective to implement a simple language, designed to make development for TinyOS easier. The design and implementation of this language is discussed in Chapter 5.

The final objective of the project consisted of using the port of TinyOS and the new language to create some demonstration applications, thus showing the work undertaken could be used in a real world project. The sample networks are described in Chapter 6.

Chapter 7 discusses further work that could be carried out to extend the project but was not undertaken due to time restrictions or because it was beyond the scope of the project.

1.5 Aims and Objectives

- Port TinyOS to the Amulet2e mote, implementing the required functionality to allow the operating system to run on the hardware.
- Design and implement a small language to simplify development for TinyOS.
- Create some simple WSN applications to use and test the implemented functionality and new language.

1.6 Literature Review

The Amulet2e datasheet[2] provides invaluable information about the Amulet2e processor, such as register addresses and timings. Documentation relating to the Amulet2e mote, however, is non-existent, and information relating to its functionality is derived from existing code written for it.

The vast majority of the literature relating to TinyOS is contained within the TinyOS Documentation Wiki[3], and the TinyOS Extension Proposals. The nesC language reference[4] provides the majority of the information relating to nesC, including syntax and example programs.

1.6.1 TinyOS Extension Proposals

A TinyOS Extension Proposal (TEP) is a formal document that defines various standards and recommendations relating to TinyOS. There are currently four types of TEPs: Best current practice; Informational; Documentary; and Experimental. TEPs 1-100 are best current practice, TEPs 100+ can be of any other type, but are predominantly informational.

The structure and types of TEPs are defined by TEP1[5]. This document also defines the keywords that specify the level of compliance to which implementers must adhere to a TEP. Of particular note are MUST and MUST NOT, as these specify the basic requirements that any implementer has to meet to ensure compatibility with the rest of TinyOS.

These documents make up the vast majority of technical literature available relating to TinyOS, and are referred to frequently in this report.
Chapter 2

TinyOS and nesC

2.1 Introduction

This chapter introduces TinyOS and its associated language, nesC. It details the underlying architecture of TinyOS and the toolchain required for creating TinyOS program.

2.2 TinyOS

2.2.1 Introduction to TinyOS

TinyOS is an open source operating system designed for WSNs [6]. It originated as a research project at the University of California, Berkley, and has grown to become the de facto operating system for WSNs, with development aided by many other universities as well as companies such as Intel and Crossbow technologies[7].

2.2.2 Programs

TinyOS programs are statically linked against the operating system to create a single executable program that runs on the mote[8]. Only one program can be linked at a time and thus the executable created becomes a specialist operating system capable of performing a specific task. In this way, TinyOS is less an operating system and more a library programs can link against to create operating systems. This is a somewhat academic difference however, and from a programmers perspective can be safely ignored.

TinyOS uses a component based architecture, with functionality separated and grouped into different components[8]. Only the components implementing the required functionality of the program are included in the compiled executable. In this way the code size of the compiled programs are kept to a minimum: an essential feature for memory constrained devices such as motes.

TinyOS programs are primarily event driven: the majority of program code is executed based on external events, with no execution occurring between them[9]. This means that between events
the operating system can put the mote into a low power state, saving battery power and extending
the lifetime of the mote in the field.

![Diagram of processing unit and radio](image)

**Figure 2.1:** The processing unit may sleep while the radio is transmitting to save power

Potentially long running tasks, such as I/O or radio transmission, also make use of events in a
so called split phase model[8]. When a long running operation is invoked by a program, the call to
do so returns nearly instantaneously. Then, when the operation has finished, an event is generated
to inform the program the operation has completed. This can save power as many motes have
hardware which can run independently of the processor. An example is transmitting over the radio:
the processor can be put to sleep while the radio transmits the packet, and woken up again when
the transmission is finished (shown in figure 2.1)[10].

To support these features TinyOS programs are written in a specific language, nesC, which has
built in support for components and events.

### 2.3 nesC

#### 2.3.1 Introduction to nesC

nesC (pronounced NES-see[9]) is a programming language designed specifically for the domain of
wireless sensor networks. It has been developed alongside TinyOS, which is implemented entirely
in nesC. It is based on C but with additions to support its use in a sensor network environment[9].
Rather than functions, code resides in either a command, an event or a task.

A command is equivalent to a function in C, and contains a section of code which may be called
by other components. Commands are called directly, much like C functions, but must be prefixed
with the call keyword.
nesC is an event driven language[9], and as such allows for code to be executed based on external events. This code resides in an event handler, which can be raised by the use of the signal keyword.

Tasks allow for deferred execution of code. They may be added to the execution scheduler with the post keyword.

2.3.2 Components andWirings

The biggest obvious difference between nesC and other programming languages is that the basic building block of a program is a component[9].

Each component consists of some code and a set of inputs and outputs, specified via interfaces, to which other components may be wired. Wiring is the process of matching interfaces to components that implement the required functionality. A component is said to use an interface if it relies on another component to provide the functionality, and is said to provide an interface if it implements functionality other components rely on[11].

Components communicate by calling commands on interfaces they use and raising events on interfaces which they provide. Thus a wiring is a bi-directional contract: a component can call any commands on an interface it uses but must also implement any events the interface may signal (as shown in figure 2.2)[9].

Usually a component is split into two parts: a module and a configuration[11]. The module contains the code and specifies the interfaces it uses and provides. A configuration file is then created to wire some of the interfaces with defaults. This allows a component to hard wire some interfaces to components, while allowing others to be wired depending on usage (shown in figure 2.3). When wiring to a component it is usually the configuration that is being wired to.

Note that a module does not have to have a configuration and other components can wire directly to a module if they wish to. This is rare, however, as most modules will have other components that they rely on to work and thus want to automatically wire to them.

The ability to wire to modules directly brings a secondary consideration of component visibility. As nesC has no support for access modifiers such as private or public a component author has no
way of specifying that the configuration should be wired to and not the module. To circumvent this, nesC has a naming scheme where private modules and configurations end with the letter P and public ones end in C[11]. Thus most components consist of two files, ComponentP.nc which contains the module and ComponentC.nc which contains the configuration.

### 2.3.3 Concurrency

Code in nesC is divided into two categories: that which is reachable from an interrupt handler and that which is not. The former is known as asynchronous code and is marked as such with the async keyword[4]. This allows nesC to statically analyse the code for race conditions where an interrupt changes the state of some shared variable used by non interrupt code. To allow access to shared variables, nesC provides the atomic block statement. Any code wrapped in an atomic section will have interrupts disabled before entering and re-enabled on exit[4].

nesC also includes support for tasks, which allow for deferred execution of code[8]. When a task is posted it is added to a queue and scheduled for execution at a later date. A user can override the scheduler implementation but the default is a First in, First Out queue[8]. Tasks are particularly useful for interrupt handlers as they allow the interrupt to perform a small amount of time critical execution in an interrupt context and perform the rest of the non critical processing at a later date in a non interrupt context.

### 2.3.4 Optimisation

As nesC is designed for use in sensor networks it aims to produce programs which use as few CPU cycles and little memory as possible. Its strategy to achieve this is twofold: firstly through language features and secondly through compile time optimisations[9].
nesC encourages a reactive style of programming, in which code is executed as a response from an external input (e.g. an interrupt) rather than polling state and continually executing. This means that between events, when there is no code to execute, the CPU can be put into sleep state to conserve power rather than wasting cycles idling.

Memory allocation in nesC is entirely static and dynamic allocation is not allowed\cite{4}. This means that the memory requirements can be calculated through static analysis at compile time, allowing for aggressive optimisation of memory. The use of components allows for large sections of unused code to be discarded: if a component is not wired to anything then it is known that the code cannot be reached during execution and thus can be discarded.

Further, the nesC compiler also replaces the call sites of small functions with the body of the functions itself, a process known as inlining, to reduce the execution overhead of calling into them for only a small increase in code size\cite{4}.

\subsection*{2.3.5 Example Program}

Listings 2.1 and 2.2 show the TinyOS blink example program. BlinkP.nc (Listing 2.1) shows the module code, where the interface the module uses are specified and the events generated by them are handled. BlinkC.nc (Listing 2.2) shows the configuration file where the correct components are wired to the interfaces the module uses.
Listing 2.1: BlinkP.nc: the module code for the blink program

```nesc
module BlinkP
{
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Leds;
    uses interface Boot;
}
implementation
{
    event void Boot.booted()
    {
        call Timer0.startPeriodic( 500 );
        call Timer1.startPeriodic( 1000 );
    }
    event void Timer0.fired()
    {
        call Leds.led0Toggle();
    }
    event void Timer1.fired()
    {
        call Leds.led1Toggle();
    }
}
```

Listing 2.2: BlinkC.nc: the configuration file for the blink module

```nesc
configuration BlinkC
{
}
implementation
{
    components MainC, BlinkP, LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;

    BlinkP -> MainC.Boot;
    BlinkP.Timer0 -> Timer0;
    BlinkP.Timer1 -> Timer1;
    BlinkP.Leds -> LedsC;
}
```
2.4 TinyOS Architecture

2.4.1 Platforms, Chipsets and Sensor Boards

Each physical mote in TinyOS is represented by a platform. A platform contains the code necessary to interface TinyOS to the physical hardware, a set of support files and tools for compiling and installing programs, and optionally a second set of tools for optimisation and instrumentation for programs running on the platform (shown in figure 2.4)[12].

![Figure 2.4: TinyOS Platform Layout](image)

A chipset is similar to a platform but contains only the code for a specific part of a platform, such as the radio chip. This allows for code sharing across platforms that contain the same chips. This is particularly useful for different versions of the same motes as many of the chips are the same and the platform files simply have to wire the chipset code together in the correct way[13].

Additionally, TinyOS has the notion of sensor boards. These represent detachable boards that can be connected to a mote to extend its functionality. These are typically a collection of chipset components wired together with some specific logic providing access to them[12].

2.4.2 Hardware Abstraction Architecture

TinyOS has a three layered Hardware Abstraction Architecture (HAA), as defined in TEP 2[14]. The majority of the code for a platform consists of implementing these layers (shown in figure 2.5).

The lowest level is the Hardware Presentation Layer (HPL). This provides the lowest level access to the hardware, and provides wrappers around low level tasks, such as setting registers, with higher level software calls. Components in this layer are not expected to be used from user programs as any functionality they provide should be exposed by higher levels above it. As such there are no implementation restrictions on the HPL but it suggested it be stateless and efficient as possible, as many higher level components may call into it simultaneously.

Above the HPL is the Hardware adaptation layer (HAL). This layer contains most of the platform code, and exposes one or more components for each piece of functionality the mote exposes. TinyOS also places no restriction on the HAL layout as it will be specific for each platform. However, standard naming procedure should be followed for components as they may be used directly by user programs.

HAL level components provide advanced functionality of the platform at the expense of programs using them becoming platform specific.

The final layer is the Hardware Interface Layer (HIL). The interfaces and components in this layer are defined by TinyOS and each platform is expected to provide them. The majority of the
code in these components is simply mapping between the standard interfaces and those defined by the HAL. If a program only uses components in the HIL then it should run correctly on all TinyOS platforms without changing any code. However, as the HIL has to interface with all platforms, the functionality it provides is the lowest common denominator of all available mote hardware.

2.5 Toolchain

2.5.1 Build System

TinyOS uses a make based system for compilation[15]. Using a comprehensive set of make rules and a custom make file for each program, a user can build a program for any platform by typing `make platform` in the program directory, where `platform` is the name of the platform to build for[13].

Each platform supplies its own custom set of make rules which customise the build process for the specifics of that platform. A platform may also optionally specify a set of extras which can be appended to the make command, allowing for features such as automatic downloading to the mote after compilation.

Each platform also supplies a `.platform` file which specifies the location of its components, so that they may be found during compilation[12].
2.5.2 nesC compiler chain

Internally, the make system invokes the nesC compiler for TinyOS (NCC) with the correct options sourced from the platform specific files[16]. NCC is an extension to the nesC compiler (NESCC) which automatically sets some of the options required for TinyOS programs (shown in figure 2.6).

NESCC itself is an extension to the Gnu Compiler Collection (GCC)[17][18]. It acts as a front end to the compiler, and internally compiles the nesC into an intermediate C representation. This can then be passed to a GCC back end to generate the correct machine code for a platform. The GCC back end used can be specified via command line argument, and the make system reads this from the .platform file.
Chapter 3

The Amulet2e Sensor Mote

3.1 Introduction

This chapter introduces the Amulet2e mote (shown in figure 3.1), developed by the Advanced Processor Technologies (APT) group in the School of Computer Science at the University of Manchester[2].

Figure 3.1: The Amulet2e Sensor Mote[19]
3.2 Features

At the core of the Amulet2e mote is an Amulet2e processor (shown in figure 3.2), developed by the same group as the mote. It is an asynchronous microprocessor and conforms to the ARM v4G[2] instruction set.

![Amulet2e Hardware Layout](image)

Figure 3.2: Amulet2e Hardware Layout[20]

It has two sensors for measuring the external environment: a Light Dependant Resistor (LDR); and a digital thermometer. There is also a voltage reference, at a nominal value of 1.25 V, and two pins to allow for external input to measured. For simple output it has a single Light Emitting Diode (LED), a piezoelectric buzzer, and a detachable 2x16 Liquid Crystal Display (LCD) screen. 32 kilobytes of non-volatile Electronically Erasable (EE) memory is also provided, allowing for data to be stored on the motes even if they run out of power. For mote-to-mote communication it uses a small, custom-built radio. Mote-to-PC communication works via a Universal Asynchronous Receiver and Transmitter (UART) and a detachable board providing RS232 Serial communication (shown in figure 3.3).

The Amulet2e mote runs the ARM ANGEL debug monitor as part of its firmware. This allows communication between the mote and a host PC, allowing programs to be debugged on the computer as they run on the board.

3.3 Existing Code

The Amulet2e mote comes with a set of low level drivers written by Steve Temple also of the APT group. The drivers, written in a mixture of C and assembler, provide access to the underlying
3.4. TOOLCHAIN

The GNUARM suite of tools are used to compile and download programs to the mote. These are ARM specific versions of the standard GNU GCC suite[21]. Version 3.4.3 is used as this is known to communicate correctly with the motes, although other, later, versions may also function correctly.

Programs written using the original drivers are compiled using the arm-elf-gcc compiler. This is a GCC backend that outputs ARM executables in the ELF format[21]. The program is statically linked against the drivers to create a single executable (shown in figure 3.4).

The compiled program is then downloaded onto the mote using arm-elf-gdb. This is an ARM specific version of the GNU Project Debugger (GDB)[22]. Whilst primarily used for debugging programs, this tool can leverage the Remote Debug Interface (RDI) libraries[23] to communicate with the ANGEL debug monitor installed on the mote. This allows for the program to be downloaded and run.
Figure 3.4: Original Driver Toolchain
Chapter 4

Porting TinyOS

4.1 Introduction

Porting TinyOS was the primary aim of the project. This chapter describes the main challenges associated with this process and how they were solved or how workarounds were developed for them.

To port TinyOS, a new platform was created, known as Amulet2, that allows TinyOS to interact with the hardware on the Amulet2e mote. The vast majority of the code involves implementing the Hardware Abstraction Architecture (HAA), each layer of which is described in detail. An overview of the finished HAA is given in figure 4.1. The remainder of the platform code is related to the infrastructure of the platform, and is very similar to other platforms, so has been omitted for brevity.

An automated toolchain was also created, allowing the platform to plug into the TinyOS build system. This contains extra functionality from other platforms which is also discussed.
* The temperature, voltage and light sensors all provide the HIL level \texttt{Read} interface.

Figure 4.1: The Amulet2 Platform Architecture
4.2 Hardware Presentation Layer

4.2.1 Component Layout

The Hardware Presentation Layer is implemented in a single component, the Hpl component, and is responsible for the low level interaction between the OS and the hardware. The primary role of the Hpl component is to handle interrupts and notify registered components that an interrupt has occurred.

4.2.2 Interrupt Handling

The Amulet2e mote is only able to generate interrupts based on timer events[2]. A program can set the value of the timer comparison (CMP) register and when the timer (TIM) register becomes equal to this value an interrupt will be generated. The CMP register can then be set again, to generate the next interrupt. As such, all other external processes must be polled on this interrupt to see if their state has changed.

The Hpl component is responsible for setting the CMP register and handling the generated interrupts. To allow other components to receive notification of interrupts, the Hpl component provides a command, Hpl.registerHandler(void (*funcPtr)()), which allows a component to register a function to be called when an interrupt occurs (as in figure 4.2).

![Diagram showing the process of registering for interrupt notification.]

Figure 4.2: Component registering for interrupt notification

In this way, each registered component is notified that it should poll the state of any external processes it is responsible for. As this registration is dynamic and occurs at component initialisation, any components that are not used by a particular program will not register a function, and thus not waste resources polling when the outcome will not be used.
The main problem with the registration-based handling is the performance impact associated with calling multiple functions from the interrupt handler. As the handler has no way of knowing if the component being notified has any need to perform any work, each handler must be called every time. Further, if several handlers are registered and each performs complex processing, a noticeable delay can be introduced. As interrupt handling is extremely time critical this is unacceptable. If the delay is longer than the time between interrupts then the CMP register will be set to a time before the handler finishes and no further interrupts will be generated (shown in figure 4.3). Eventually the timer register will wrap around and generate an interrupt, but as the timer is a 16 bit counter and runs at 32 kHz this will introduce a delay of approximately 2 seconds.

![Diagram of timer values and interrupt handling](image)

The interrupt handler in normal operation: at timer value 1, an interrupt is generated and handled; the CMP register is set to generate another interrupt after SLEEP_INTERVAL, which in this case is 16. Thus the interrupts are generated and handled correctly.

The effect of the handler taking too long to process: the interrupt at 1 is handled, and the CMP register is set to generate another interrupt at 17; however, the interrupt handler is still running at 17 and the interrupt is missed; the timer will have to roll over back to zero before becoming 17 again for the interrupt to be handled.

Figure 4.3: Effect of the interrupt handler taking too long

Preventing this delay has two consequences. Firstly, each handler registered must try and evaluate if it has any work to do as quickly as possible and return as soon as it knows it does not. The second consequence is much larger and requires that the time between interrupts be several times larger than that used in the original drivers. This has further consequences for radio communications,
discussed later. Based on empirical testing of interrupt times against numbers of registered handlers a time of 16 ticks between interrupts was decided upon (compared with 2 ticks in the original drivers). This allows for all handlers to complete but is still quick enough for the individual handlers to recognise external events. This time is defined by the global SLEEP_INTERVAL constant allowing components to adjust their behaviour without hard coding in the interval.

Unfortunately, these tradeoffs are a consequence of the design of TinyOS and the different way in which the Amulet2e mote works compared with other motes. These tradeoffs appear to be the most satisfactory way of running TinyOS without fundamentally changing its architecture.

4.2.3 Changing Power States

TinyOS has a mechanism to put the processor into a lower power or sleep state if there are currently no tasks to execute\[24\]. This is exposed through the TinyOS McuSleep interface defined in the HIL. Figure 4.4 shows how this is implemented on the Amulet2 platform.

![Figure 4.4: McuSleep puts the processor to sleep until the next interrupt](image)

On some platforms, where the processor supports multiple power levels this can be a complicated process\[24\]. The Amulet2e processor however has just two such states, running and sleeping\[2\]. This allows the McuSleep component to be particularly simple, if it is requested to change power state it will always put the processor to sleep. The code to perform this is exposed via the HPL as it requires assembly code and affects the functionality of the interrupt handler.

To place the Amulet2e processor into a sleep state you simply have to enter into an infinite loop, which in assembler is the \texttt{B} (branch to self) instruction\[2\]. The processor will detect this and enter a sleep state.

On other mote platforms, it is expected that an interrupt will wake the system and execution will continue as before\[24\], but this is not the case on the Amulet2e mote. Due to this, the interrupt handler must be aware of whether or not the processor was in a sleep state when it executed and act accordingly. This is determined by a flag, set when the sleep command is called. Because the interrupt handler function is defined as such by a compiler attribute, the compiler will automatically
add in the required assembler to save and restore the registers at the beginning and end of the function respectively, and cause the flow of control to return to the point where the interrupt occurred. This causes a problem when the processor was asleep, as it will return to the branch to self instruction and go back to sleep rather than returning control to TinyOS. As such, when the sleep flag is set the interrupt handler will manipulate the saved register stack so that execution returns to one instruction after where the interrupt occurred, which in the case of the sleep function is a no operation instruction. This allows the sleep function to return and normal execution continue.

Waking up on an interrupt makes sense on other platforms as it means there is work to be done, however, on the Amulet2e mote this is not necessarily the case due to the use of polling. As it is not known if there is work or not it is always assumed there is, and if there is not the scheduler will put the mote to sleep again, but some power has been wasted temporarily returning control to the scheduler. This is another unfortunate tradeoff that appears to be required to run TinyOS correctly on the mote.

4.3 Hardware Adaptation Layer

This section discusses some of the components that make up the HAL. Due to the large number of components, only those that are particularly interesting or posed a significant challenge in implementing are discussed.

4.3.1 Component Layout

Most components of the Amulet2e HAL follow a standard naming and location scheme. Each component is placed in a subdirectory based upon the functionality it represents. For example, all components related to radio communications are placed in a folder called Radio. As well as making the code more organised it also has an effect on the documentation system discussed later in this chapter.

Each component is defined by three parts: an interface; a configuration and a module. In the case of the radio component and following the standard nesC naming convention this gives: Radio.nc, which defines the Radio interface; RadioP.nc, a module which provides the code for this interface; and RadioC.nc the configuration that hard wires any needed components to the module and provides the Radio interface the module exposes.

4.3.2 Radio Communication

The radio component provides a simple interface for sending and receiving streams of bytes over the radio. It is designed to be very low level and error detection and correction are left for implementation in a higher level of the TinyOS network stack. Both the send and receive functions are split phase and so have events to notify the system the operation has completed.

The radio on the Amulet2e mote works on a synchronous basis. In the original drivers there is a function to receive data which will block until data is received or a specified timeout occurs. This creates a challenge when porting to TinyOS as it expects a radio to raise an interrupt when a message has been received[25]. This means that the component must emulate this event behaviour
by continually polling in the background. Further, the radio cannot send and receive at the same time, so these must orchestrated to make it seem as if it can.

```python
if sending then
    continue_send()
else
    if receiving then
        continue_receive()
    else
        if send_pending then
            sending ← true
            start_send()
        else
            receiving ← true
            receive_start()
        end if
    end if
end if
end if
```

Figure 4.5: Send and Receive orchestration for the radio component

Figure 4.5 shows the implementation of the orchestration algorithm. Preference is given to sending packets, and a receive will only be initiated when the radio is not currently in use. The component registers for interrupt notification on initialisation, and uses this algorithm to start and update two internal state machines. These state machines, shown in figures 4.6 and 4.7, are used to handle sending and receiving. They are ports from the original drivers where they were originally written in assembler.

As mentioned in section 4.2.2 the rate at which interrupts are generated is less than that in the original drivers. This means that times in the state machines have to be adjusted to be slower, as the delay between each update of the state machine is increased. This essentially means that each
1. Enable receive
2. Wait for first preamble
3. Wait for second preamble
4. Wait for padding bit. If '1' go to 5, if consecutive '0' go to 7
5. Read byte, one bit at a time
6. Store byte, return to 4
7. Disable receive

Figure 4.7: State machine for receive

bit sent by the radio is sent for 8 times longer (or sent for the same amount of time 8 times in succession), but because the reading is also slowed by the same factor the duplication is cancelled out and only a single bit is read. The practical outcome of this is an overall decrease in the data transmission rate. Whilst somewhat unsatisfactory, it is the only way that has been found to get the radio running correctly under TinyOS.

4.3.3 Timer

The timer is one of the most important aspects of TinyOS and as such has very strict rules governing its functionality, set out in TEP102[26]. Whilst the internals of this component are actually relatively simple, adherence to these rules makes the implementation of this component slightly different from the others.

The timer component represents a millisecond precision timer, which can be set to fire once after an elapsed amount of time, or on a repeating schedule. The component keeps track of time by keeping an internal counter. When it is initialised it registers for interrupt notifications and increases the counter by the SLEEP INTERVAL constant when notified. In this way, each interrupt brings the counter into sync with the hardware time.

The internal clock is driven by a 32 kHz timer, so every 32 ticks represent one millisecond. On each interrupt the counter checks to see if the requested time has been reached, and if so posts a task to notify the rest of the system. In this way the notification process occurs in a non-interrupt context. If the timer is set to repeat the notification flag is then cleared and the next time is set, otherwise it is disabled.

A TinyOS platform may optionally provide both 32 kHz and microsecond precision timers in addition to the required millisecond one[26]. As the Amulet2e hardware timer runs at 32 kHz a microsecond timer can not be provided, but in terms of hardware a 32 kHz one is theoretically possible. Unfortunately, because interrupts only occur at intervals of SLEEP INTERVAL, it would only be possible to update the timer in increments of this constant, thus rendering its improved accuracy useless. For this reason it is not provided.
4.3.4 Serial Communication

The serial component provides low level sending and receiving of bytes over the UART, much in the same way the radio component does. Again, send and receive are split phase.

Interrupts are once again a problem, but rather than occurring too little they occur too often. TinyOS expects the serial component to notify when data is received in one of two ways: either notifying every time a byte is received, or once a specified buffer is full. To handle the reception in a non-interrupt context a task must be posted when either of these happens. In single byte notification mode, the default in TinyOS[27], the overhead of posting a task every time a byte is received causes the interrupt handler to take too long and miss the next interrupt. Further, if a large amount of data is being sent the same is also true.

To avoid this problem, an internal buffer is always used to receive data, and a timer is used to check it once every 50ms. If the component is in single byte reception mode, it is emulated by repeatedly signalling a byte reception for each byte in the buffer. Otherwise the received data is copied to the reception buffer and handled as usual.

To avoid the sending of large amounts of data slowing the interrupt handler, sending is also performed from the timer in a non interrupt context. Rather than sending a single byte each interrupt, the bytes are queued and sent in batches of up to 100 (the equivalent number of interrupts between each send). This works as the UART buffers data so from the computers point of view the data is received no differently.

The serial component currently only runs at a baud rate of 9600 bps, but the hardware supports several other faster speeds. The ability to select different baud rates is left to further work.

4.3.5 I2C Bus

The I2C component allows read and write access to the I2C bus. Unlike the serial and radio components, the functions are synchronous.

In the original drivers the I2C bus access was written in assembler and so has been ported to nesC. However, it still includes several embedded assembler statements to allow for the correct delays during access. Whilst these delays are not noticeable on single reads and writes, components that use the I2C bus, especially the LCD component, tend to use multiple calls which can lead to some delay. During normal execution this is not a problem but means that these components cannot be used easily from an interrupt context.

Often programs using the original drivers insert a small delay between calls to the I2C bus. This is one area where the added overhead of TinyOS is appreciated as the latency introduced by the TinyOS calling mechanism is enough that these delays can be removed.

The I2C component is not designed to be used from user code, instead it is designed to support other components that build on it. While TinyOS does have a standard mechanism for accessing an I2C bus from user code[27], this is not provided by the Amulet2 platform as it would not provide any functionality not already exposed by other components.
4.4 Hardware Interface Layer

As in the HAL there are many components that constitute the HIL; this section describes only some of the more interesting and challenging ones.

4.4.1 Timer

The HAL level timer only allows for single timer. However the use of virtualisation at the HIL level allows for multiple timers to be used in programs (shown in figure 4.8).

![Diagram](image)

Figure 4.8: Multiple virtual timers can map onto a single hardware one

The HIL level timer component is generic. This means multiple instances of it, and thus multiple timers, can be created [4]. TinyOS provides a component called VirtualiseTimer which will map multiple software timers onto a single timer [26], while still allowing them to appear to operate independently. When a new timer component is initialised, a unique ID is generated internally which allows this mapping to take place.

The use of a generic component makes this process transparent from the user, to whom the mote appears to have multiple timers.

4.4.2 LEDs

The LEDs HAL component is very simple and simply writes to the parallel port to set the state of the LED. Mapping this into this from the interface layer is, however, more complicated.

TinyOS assumes that a mote has three LEDs, numbered 0, 1 and 2, each with their own on and off commands. This is not changeable and many of the test programs rely on three being available. This is one of the areas where the design of TinyOS appears to be lacking. A more expandable solution would be to have a query method that returns the number of available LEDs, and have the on and off methods take an argument as to which LED it applies. This pattern is used in other areas of TinyOS and its absence here is unfortunate.

To satisfy this hard coded requirement, the component maps all three LED on and off commands to the one physical LED. This allows the test programs to run, but can result in somewhat unexpected LED behaviour. For example, if LED 0 is turned on and LED 1 is turned off simultaneously the
state of the physical LED does not change. As the three LEDs are a requirement of TinyOS, this is the only way to solve this.

To alleviate the problem of unexpected behaviour a second component was also created, Leds2Lcd. This maps the three LEDs onto the last three characters of the LCD screen with a 1 and a 0 representing state. This component conforms to the same interface as the standard LEDs component and so can be swapped by a user by simply wiring to it instead. In this way, the test programs can be run and the correct behaviour observed.

### 4.4.3 Radio Communication

ActiveMessage is the bottom of the TinyOS networking stack[10]. This maps the TinyOS networking paradigm into the native radio one, which in this case is a stream of bytes.

The native TinyOS message abstraction is the `message_t` structure[28]. This represents a network packet and contains fields for packet headers and data. When sending, the radio component is responsible for turning the `message_t` structure into a stream of bytes which are passed into the HAL radio component. It registers for confirmation that the send completed, and propagates this confirmation back up the networking stack when it occurs. The same is true in reverse for reception.

Whilst this mapping is relatively simple, TinyOS also expects the component to provide several message manipulation routines[25]. These involve calculating the maximum possible size of a message, and adding and removing radio specific headers. As the Amulet2e radio simply works on a stream of bytes, and has no concept of a packet as such, the maximum size calculated is based on the TinyOS default value and the header manipulation routines are left empty.

### 4.5 Toolchain

As discussed previously, the TinyOS toolchain is based on make. The Amulet2 platform integrates with this system and provides several extras.

#### 4.5.1 Basic Compilation

TinyOS has a mechanism whereby the nesC GCC front end can pass the intermediate representation of the program to another GCC back end[16], allowing compilation for nesC to take place for any GCC supported platform. Unfortunately, the version of the GNUARM toolchain used does not support this feature and a second mechanism must be used.

The nesC compiler has the option to output the C representation of the code rather than a binary file[16]. Using this the compilation can be run in two stages, one to produce a temporary C file from nesC and a second to produce the executable program from the C file. This two stage compilation process is automated via the make system and so is transparent to the user (shown in figure 4.9).
In the conversion from nesC to C, commands and event on components become functions in the global scope. To prevent name clashes nesC prefixes the functions with the interface name and component name separated by dollar signs[17]. A command called Radio.read(byte* buffer, int length) in the module RadioP would become RadioP$Radio$read(byte* buffer, int length) in C. Unfortunately, the GNUARM toolchain is once again a problem, as it does not support dollar signs in function names. Thankfully, the nesC compiler supports an option to change the separator character used and a double underscore was chosen. Thus the command becomes RadioP__Radio__read(byte* buffer, int length). Name clashes are still possible but uncommon: if a function called Radio__read(byte* buffer, int length) was defined in the RadioP module this would clash with the above function. For this reason, double underscores should be avoided in function names.

4.5.2 Simulate

The Amulet2 platform contains some support for software simulation. The primary focus of the simulation code is to simulate interrupt generation. This is achieved by simulating the hardware timer and manually branching to the interrupt handler when required. While this is somewhat limited, it allows for testing and step through of low level areas of the system that cannot be performed on the real hardware.

To enable this simulation support the user simply has to append the simulate option to the make command when building a program. This enables simulation support in the code by defining the SIMULATE constant. The build process will then automatically launch GDB, connect to the simulation target and download the code into the simulator. The user is then able to set any required breakpoints or watches and simply has to type run to begin simulation. An example simulation is shown in figure 4.10.
4.5.3 Download

The download option allows a user to automatically download the program to an Amulet2e mote after compilation. This saves time whilst developing, as the process of downloading consists of multiple steps (see figure 4.11).

After compilation the download option will create a temporary command file, containing the necessary steps to perform the installation of the program onto the mote. The mote is assumed to be located at /dev/ttyS0 although this can be overridden using the TARGET environment variable. The user is then prompted to reset the mote at this location, and the script waits for user confirmation before continuing. Whilst resetting the mote is not always necessary, failing to do so can cause the download to fail, so the user is always prompted to. After confirmation, GDB is launched and provided with the command file. This causes the correct baud rate to be set and a link established to the mote. The program is then downloaded to the mote, and once finished the user is presented with a GDB command prompt. Much like simulation, the user can then set any required breakpoints and types run to begin execution.
4.5.4 Docs

The final supported option is docs. This is a standard TinyOS option that will generate Javadoc style documentation known as nesdoc for the platform[29]. Information is taken from the signatures of components and their associated comments[30]. Namespaces for each component are also generated based on their location in the file system. In this way related functionality contained in e.g. the radio folder, will all be placed in the `platforms.Amulet2.Radio` namespace.

Nesdoc also produces interactive wiring diagrams showing how the various components are wired to each other, allowing a user to progressively explore the levels of the system starting from the top[11].

4.6 Result

The porting phase of the project was overall very successful. A complete implementation of a TinyOS platform was produced, and all the required components successfully implemented.

As well as producing the required components for the platform, a build system was also produced which allows for automated compilation and downloading. Simulation support was also built into the platform, allowing a developer to debug the lower level aspects of the platform which are not accessible from the physical hardware.
Chapter 5

Hench

5.1 Introduction

This chapter introduces a new language, Hench, which aims to make programming on TinyOS simpler, especially for developers new to TinyOS. Hench is a recursive acronym for Heck of a lot Easier than nesC: Hench.

5.2 The need for a new language

TinyOS programs are written in nesC. While the design of the language is suited to sensor network programming, and ideal for implementing TinyOS, writing programs in it can be somewhat cumbersome. This is especially true for new developers to TinyOS, who have to learn not just a new language but potentially a new programming paradigm also.

To create a new program in nesC a developer has to create two files: a module file for the main code of the program; and a secondary configuration file to wire the required components to the module[8]. This slows initial development and requires the developer to know which TinyOS components implement the interfaces being used in the program.

Whilst porting TinyOS to the Amulet2e mote it was observed that the overwhelming majority of the time the same components were being wired to the same interfaces. Thus if this step could be automated by the language the initial development time would be decreased, and the developer would not need to know which components are required, just the interfaces they wish to use.

5.3 Design

It was decided to base the new language on C, with extensions for specifying the TinyOS interfaces used by the program. C was chosen as the base language as many developers are already familiar with it, and as nesC is itself a C dialect, can be mapped to nesC relatively simply.

Rather than directly generating machine code it was decided to generate nesC code instead. This greatly simplifies the compilation process as a large amount of the complexity of compilation can be offloaded to the nesC compiler.
It should also be noted that Hench is not designed to replace nesC but complement it. Hench is only designed for creating programs that run on TinyOS, and so does not contain all the functionality available in nesC. The practical outcome of this is Hench can be used to program top level components (i.e. TinyOS programs), but not lower level ones which other components depend on.

5.3.1 Specifying the Interfaces Used

To specify that an interface is used by a program a new keyword, import, was added. This has four forms of use shown in listings 5.1 to 5.4.

While the interfaces used in the program will often map to a known component, this is not always the case. Some interfaces may map to multiple components. An example of this is the read interface: each sensor provides this interface as a standard way of reading data from it, so there can be no default mapping. A developer may also want to override the default component wiring, such as substituting the standard LED component for the Leds2Lcd component on the Amulet2 platform. For this reason the import statement has the ability to override the default mapping using the '=' operator (Listing 5.2). Further a program may use more than one instance of the same interface, so import supports the ability to alias interfaces with a different name using the 'as' operator (Listing 5.3). Component overriding and alias may also be used together in the same import statement (Listing 5.4).

Listing 5.1: Basic import statement specifying the program uses the Leds interface

```
import Leds;
```

Listing 5.2: Import statement overriding the default Leds wiring

```
import Leds = Leds2Lcd;
```

Listing 5.3: Import statement with an alias

```
import Leds as MyLedsAlias;
```

Listing 5.4: Import statement with overriding and an alias

```
import Leds as MyLedsAlias = Leds2Lcd ;
```
5.3.2 Event Handling

In nesC, a component wiring is bidirectional: a component A using another component B can call commands on B, but must also handle any events B may signal. In practice it is often the case that B signals events that A does not need to listen to, but must still handle.

To remedy this situation, Hench introduces a new language construct ‘\=>’ which allows for mapping of events to event handlers in the program. The event handler mapped must have the same arguments and return type as is specified by the original event. If the program does not map an event, a default, empty, event handler is generated automatically.

In this way, the program only contains code directly related to the program, and not empty event handlers inherited from other components. It also allows the developer to rename event handlers to be more descriptive: rather than naming them by what they handle, they can be named by the functionality they perform. An example of event handler mapping is shown in listing 5.5.

Listing 5.5: Mapping the timer fired event to a custom handler

```
Timer.TimerFired \=> toggleLed;
```

5.3.3 Other Notes

As in C, the main program code resides in the `main` method. Functions in the program follow the same calling convention as C, i.e. the `call` keyword from nesC is not necessary. To call commands in other modules, the command name is simply prefixed with the interface name or its alias.

The `post` and `task` keywords are retained from nesC, and perform the same functionality. Thus if a function is marked with the `task` keyword, another function may schedule it for deferred execution with the `post` keyword.

Finally, the file extension `.h` was chosen for Hench files. While this is also used for C header files, it has the benefit that many editors assume C syntax when opening it, allowing for the vast majority of the Hench code to be syntax highlighted when supported.

5.3.4 Example Program

Listing 5.6 shows a port of the TinyOS blink program (shown in listing 2.1) to Hench. It shows importing the required interfaces, aliasing the multiple timer instances, and mapping the timer fired events.
```java
import Timer<TMilli> as Timer0;
import Timer<TMilli> as Timer1;
import Leds;

Timer0.fired => ToggleLed0;
Timer1.fired => ToggleLed1;

void main()
{
    Timer0.startPeriodic( 500 );
    Timer1.startPeriodic( 1000 );
}

void ToggleLed0()
{
    Leds.led0Toggle();
}

void ToggleLed1()
{
    Leds.led1Toggle();
}
```

Listing 5.6: Hench version of the TinyOS blink example program

### 5.4 Implementation

The Hench toolchain was developed primarily using the ANTLR Parser Generator, developed by Terrance Parr of the University of San Francisco[31]. This tool takes a grammar document specifying a language, and generates a lexer and parser for it, allowing a compiler for the language to be created.

#### 5.4.1 The Hench Grammar

As Hench is based on C with extensions, it was decided to extend an existing C grammar, rather than constructing one from scratch entirely. An existing ANTLR grammar for C, written by Terrance Parr, was used as the basis for the Hench grammar[32].

The extensions to the grammar add the support to recognise the import and mapping statements, as well as adding the task and post keywords. Figure 5.1 shows the ANTLR syntax diagram of the event handler grammar statement, generated by the ANTLRWorks tool*. Figure 5.2 shows the complete grammar extensions as a graph of recognition rules.

*ANTLRWorks is an integrated development environment for creating ANTLR Grammars[33]
5.4. IMPLEMENTATION

Figure 5.1: Hench event handler assignment grammar syntax

Figure 5.2: Complete Hench grammar extensions as a graph of recognition rules

When run, the parser generates an Abstract Syntax Tree (AST), which can be traversed by the compiler to generate the required output. Internally, it marks each AST node as either a C statement or a Hench extension, allowing for easy recognition in the compiler.

5.4.2 The Compiler

The Compiler was split into two phases: the first phase generates the nesC module file and a list of used interfaces; The second phase generates a configuration file specific for a TinyOS platform. In this way Hench can be platform agnostic, and generate programs for any TinyOS platform.

Module Generation

Module generation is handled by the Hench.Compiler.exe program (shown in figure 5.3). This is implemented in C# and uses the ANTLR generated lexer and parser to read in Hench files.
The AST created by the parser is traversed and the internal labels are used to distinguish Hench code from C. When an import statement is encountered the compiler notes the interface used (and any overrides and aliases) and discards the line. Similarly, when event handlers are encountered the mapping is recorded and the line discarded.

When a C statement is encountered the compiler performs several actions. The main method is automatically turned into the `Boot.booted` event handler, and the `Boot` interface is automatically added to the list of used interfaces. Any calls to external component commands are prefixed with the `call` keyword. Functions that have been mapped to event handlers are turned into the events they are handling. The C statement is then added to an internal AST that will form the final module.

After traversal, the compiler creates a nesC module with the interfaces that were imported transformed into `uses` statements. The newly generated AST is then used to output the rest of the code as nesC into this module. The list of interfaces and associated overrides and aliases are written to an intermediate XML file for use by the configuration generator.

Due to the complexity of building a compiler, and the time constraints of the project, the ability to generate default events has been left to further work. To correctly generate the default events the compiler would need to also parse the nesC component where the event is declared so that the correct method signature can be used, which was beyond the scope of this project.

One further caveat of the compiler comes from the fact that C is not a context free language. For example, when declaring a type, it needs to be remembered later on that this type exists so it can be parsed properly. As TinyOS uses many non standard types, this information needs to be known by the parser so that it can identify them as such. Rather than hard coding the types into the grammar, an external file is used to list the types in TinyOS. This is loaded before parsing and the types noted, allowing them to be identified correctly by the parser when encountered. Ideally this list would be generated automatically by parsing the TinyOS source, but this is again beyond the scope of the project.
Configuration Generator

The second phase of compilation consists of generating the configuration file for the module. This is handled by Hench.ConfigurationGenerator.exe also implemented in C#. This stage is platform specific and the configuration generator accepts an argument specifying the platform to generate for.

The configuration generator reads the XML file generated by the compiler to match the correct components to the interfaces used by the module. This matching is achieved by reading an XML mapping file, which lists which components implement which interfaces. Because each platform uses both standard TinyOS components and its own custom ones, the mapping files operate on several layers (as shown in figure 5.4). There is a TinyOS mapping which defines the standard TinyOS components, while each platform also has its own mapping file. This allows a platform to override the TinyOS defaults as well as map its own custom components to interfaces. There is also a final layer of component overriding from the generated list of interfaces, where the module may specify a specific component for an interface manually.

From these mapping files and the generated list of interfaces the configuration creator can create a nesC configuration file, with the correct components mapped to the module interfaces.
5.4.3 HenchC Script

The final piece of the Hench toolchain is the HenchC script. This is a bash shell script that automates the process of building Hench programs. It works on both Linux and Cygwin by checking environment variables to adjust paths to the correct format and invoke the processes in the correct way (on Linux it invokes the C# based tools with mono†).

As arguments it accepts the name of the Hench file to compile (without the .h extension) as well as a set of arguments to pass to the TinyOS build system. An example invocation of this script is shown in figure 5.6. The script is currently hard-coded to select the Amulet2 platform when executing the configuration creator, but could be easily changed to accept a platform argument to pass to the configuration creator. To enable the automated handover to the TinyOS build system the script also creates a makefile. This contains the required information about the generated nesC files, which allows the TinyOS build system to take over.

†Mono is an open source implementation of the .NET framework for Linux
Overall the Hench language was very successful. Several of the TinyOS example programs were successfully ported with little effort, and no shortcomings were identified.

The language was used to implement the demonstration applications shown in chapter ?? and the overall development process was positive. By having the entire program code in a single file, it becomes much more obvious what the program does, as well as making changing the components used in a program much easier.

Further, as it generates nesC code, it can be used as a learning tool for those beginning development on TinyOS. By initially writing programs in Hench, which due to its similarity to C is easy to learn, a developer can see how these programs map into nesC, thus aiding the learning of nesC and its programming model.

One area where the language does not entirely meet the design goals is in default event generation. While the language supports event mapping, the compiler does not automatically map missing events to default handlers due to time restrictions.

Due to the success of this phase, it is planned to release the language and toolchain to the wider TinyOS community. The source code for the compiler will also be uploaded to the TinyOS source repository, hopefully prompting further development.
6.1 Introduction

This chapter introduces some applications written to test and demonstrate the features of the Amulet2 TinyOS platform and the Hench language. It also introduces some simple experiments, exploring the performance of the Amulet2e mote when running an example sensor network program.

6.2 Amulet2 Tester

The Amulet2 Tester application (shown in figure 6.1) is a PC application designed to test the functionality of the Amulet2 Platform.

Written in C#, it communicates with a TinyOS program, written in Hench, over a serial port. It reads the values of the sensors on the board, and allows a user to toggle the LED and write a message to the LCD screen. It also enables a user to send a message over the radio. If a second mote, running the same software, is connected to a second instance of the application it will receive any messages sent over the radio and display them on screen.

In this way, the majority of the components that make up the Amulet2 platform can be tested for correct functionality on a range of motes. Further, the mote program only uses the HIL level components, so it is also tested that the various layers of the HAA communicate correctly.

To communicate with the standard TinyOS serial libraries used in the mote application, the tester application must understand the TinyOS serial packet format. While this is a relatively simple format, it also includes a Cyclic Redundancy Check (CRC) of the data[34]. The specific CRC used is CRC16-CCITT. Thus when sending data to the mote the application has to generate this CRC, and an existing implementation of CRC16-CCITT in C# was used to do so.[35]
CHAPTER 6. DEMONSTRATION APPLICATIONS AND SIMPLE EXPERIMENTS

Figure 6.1: The Amulet2 Tester Application
6.3 Example Sensor Network

For further testing of the motes a simple sensor network was created, with an associated PC program to display the data collected.

It was chosen to collect the light sensor readings from each mote in the network as these can easily be changed (e.g. by placing a hand over the sensor). It can then be checked that the updated reading is correctly propagated through the network.

To enable this network, three TinyOS programs were written in Hench. The first of these was Dissem. This program periodically reads the value of the light sensor and sends a packet over the radio with the sensor value and an ID unique to the physical mote. The second program was BaseDissem. This program listens for packets from Dissem on the radio and upon reception writes the sensor value and mote ID to the serial port. A final program Relay was also created to test the motes ability to propagate packets through the network. It simply listens for a packet and rebroadcasts it upon reception.

![Diagram of the sensor network](image)

Figure 6.2: How the three programs work together

The Amulet2 data collector (shown in figure 6.3) is a PC application for displaying the data from the example network in real time. It communicates with the BaseDissem program and updates the UI as new sensor data is retrieved. If a new node ID is encountered (i.e. an extra mote is added to the network) it will automatically add it to the UI.

Unfortunately due to logistical issues regarding battery packs, it was not possible to create a network larger than three nodes during this testing. Using three nodes it was possible to test the network in two configurations: one with two Dissem nodes communicating with a BaseDissem node; and one with a single Dissem node communicating to a BaseDissem node via a Relay node. In both of these configurations the network functioned correctly.
Figure 6.3: The Amulet2 Data collector program
6.4 Network Experiments

The final network setup was designed to test the battery life of the motes and calculate the error rate of radio communication.

6.4.1 Battery Life

A test was designed to discover how long an Amulet2e mote, running TinyOS, would function while running under battery power. A program was created to simulate normal operating conditions of the mote, in which it samples the sensors and broadcasts a packet on the radio once a second. The mote was also connected to a computer which recorded the voltage of the batteries whilst the program was running.

![A graph of recorded input voltage against time](image)

Figure 6.4: A graph of recorded input voltage against time

When setting up the battery life test, it was discovered that the radio on the Amulet2e mote only functions correctly under a small range of input voltages. For this reason a voltage regulator was used to keep the input voltage of the mote at the required value for as long as possible.

As the graph in figure 6.4 shows, the voltage stays at around 3.3V for a long period then relatively quickly drops as the batteries begin to run out. This is caused by the voltage regulator itself needing a certain amount of power to keep the input voltage constant. As the batteries begin to drain, and the voltage regulator begins to fail, the input voltage begins to rapidly fall and the mote eventually stops responding.

The mote lasted for approximately 2 days and 7 hours in total before it stopped responding.

6.4.2 Radio Errors

While the battery tester program was running, a second mote was set up to receive the packets broadcast by the battery tester and calculate the packet loss of the Amulet2e radio. The packet sent by the battery tester contained an integer which was incremented on each send, thus if there
is break in the sequence received by the second mote it can be easily seen that a packet did not transmit successfully.

![Graph showing cumulative packet loss against time](image)

Figure 6.5: A graph showing cumulative packet loss against time

As the graph in figure 6.5 shows, the packet loss rate is roughly linear for most of the test, until much later when it begins to rapidly increase. This rapid increase coincides with the voltage of the battery tester mote dropping. As mentioned in the previous section the radio on the Amulet2e mote only operates correctly over a very small range of voltages, so this increase in packet loss as the voltage drops was expected.

The graph also contains two sections where the loss rate increases at a greater rate. In both of these sections the increase occurs for approximately an hour, and both periods are approximately 24 hours apart. These seems to indicate that it is an unidentified external effect causing the increase in loss, not a hardware or software fault.

Taking one of the linear sections as being indicative of normal operation, the average packet loss rate can be calculated at approximately 2.5 packets per hour.

### 6.4.3 Error Rates at Different Distances

A second set of experiments were also conducted to measure the error rate of data contained within successfully transmitted packets, and how it varies with transmission distance.

Packets with a known data string were sent from one mote to another, where it was recorded if the data received matched what was expected. This was repeated for varying distances between the two motes.

Unfortunately, the results obtained from these tests proved inconclusive, although it is believed the methodology was sound. Due to time restrictions they were not able to be repeated.
7.1 Introduction

This chapter details work that could be carried out to enhance or extend the project, but was not undertaken due to time constraints or because it was beyond the scope of the project. This is not an exhaustive list, but suggests some areas that would directly add to the project.

7.2 Improvements to Hench

One of the major areas of the project that could be significantly improved with further development is the Hench language.

One of the design goals for Hench was to automatically generate event handlers for events a program did not use. To achieve this, the compiler would have to parse the nesC code where the event is originally declared so that the event signature can be deduced.

If support for nesC parsing was added to the compiler then the need to list nesC types in an external file could also be removed, as these could be identified at the same time, making the overall compilation process more robust.

Equally, the same solution to the nesC types could be applied to the event mapping, whereby each component mapping also contains a list of default handlers for the component, eliminating the need for nesC parsing. Either solution would improve the Hench language and make development using it even easier.

7.3 Serial Baud Selection

Currently the serial component only supports running at a baud rate of 9600bps, although the hardware supports speeds of up to 38400bps. To enable baud rate selection, the HAL level serial module could be configured to allow the baud rates to be selected. Several configurations could then be created which select different rates, and a developer would simply have to wire to the configuration providing the desired serial baud rate.
7.4 Debugging Support

One possible extension that would enhance development for the Amulet2 platform would be a Debug component. To debug programs the developer currently has the choice of writing out short amounts of information to the LCD screen, or sending data to a PC using the UART. A debugging component could be used to write statements and record values of variables and sensors, with a selectable option of how to output the information. It could also use the EE storage to save debugging statements when not connected to a PC to be retrieved and analysed at a later date.

TinyOS currently has some limited support for debugging in the form of the Printf library[36], but this is only implemented on a few platforms. If the debugging component could also be tied into this system, it would make debugging for TinyOS as a whole easier.

7.5 Low Battery Warning

Currently TinyOS includes no support for notifying the system when the batteries are getting low. A component could be created that monitors the voltage sensor, and raises an event when the voltage of the battery drops below a certain level. This would allow the program to save any important data to EE storage, and shut down gracefully.
Chapter 8

Conclusion

8.1 Introduction

This chapter looks back at the original project aims, and discusses how well, and to what degree they were met, as well as providing a brief evaluation of the project as a whole.

8.2 Aims and Objectives

- Port TinyOS to the Amulet2e mote, implementing the required functionality to allow the operating system to run on the hardware.

TinyOS was successfully ported, and a complete implementation of a TinyOS platform was produced for the Amulet2e mote. In addition, the implementation plugs into the TinyOS build system to allow for automated compilation, simulation, and program downloading.

- Design and implement a small language to simplify development for TinyOS.

The Hench language was created, allowing developers to create TinyOS programs using a more C-like language. Several TinyOS example programs were ported to the language successfully, and was used to create several new demonstration applications. It is planned to release the language to the wider TinyOS community.

- Create some simple WSN applications to use and test the implemented functionality and new language.

Several example networks were created, which allowed for testing of the Amulet2e TinyOS implementation, as well as allowing several experiments to be conducted using the motes.
8.3 Challenges Faced

One of the main challenges faced while undertaking this project was in mapping between how TinyOS thought a mote should work, and how the Amulet2e mote actually works. Several workarounds had to be developed to emulate certain features such as interrupt-based message reception, and how the processor enters a low power state. Often this emulation involved a trade off of lower performance, especially in the case of how quickly data can be sent over the radio. However, it is believed these trade-offs are worth the loss in performance when weighed against the benefits of being able to run TinyOS on the motes.

8.4 Project Evaluation

Overall, the project was a success. The main aims and objectives were met, and a fully functioning port of TinyOS was produced for the Amulet2e sensor mote. The added aim of producing a new language made the work carried out useful not only in the School of Computer Science, but to anyone using TinyOS. The demonstration applications also showed that the motes running TinyOS could now be used to carry out real sensor network research, and an MSc student will now be using them to investigate medium access control protocols.
Bibliography


[16] ncc - nesc compiler for tinyos. ncc man page.


