TinyWifi: Enabling Linux Platform Support in TinyOS

Bachelor Thesis

Bernhard Kirchen

RWTH Aachen University, Germany
Chair for Communication and Distributed Systems

Advisors:

Muhammad Hamad Alizai M.Sc.
Prof. Dr.-Ing. Klaus Wehrle
Prof. Dr.-Ing. Stefan Kowalewski

Registration date: 2010-05-21
Submission date: 2010-10-08

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, 20th Sep. 2010

(Bernhard Kirchen)

Abstract

We present TinyWifi, a new platform for TinyOS – the de facto standard operating system for sensornets. Although TinyOS is originally designed for use in wireless embedded sensor nodes, TinyWifi allows for compiling TinyOS applications for Linux driven devices like PCs and handhelds. Our TinyWifi (pseudo-)platform integrates seamlessly into the existing TinyOS code base and enables TinyOS users to run their applications directly on Linux without the need to alter the original software.

The primary objective of TinyWifi is to be able to use communication protocols in IEEE 802.11 wireless mesh networks originally developed for inherently similar sensornets. By featuring direct execution of unaltered TinyOS applications, especially for the purpose of evaluation, TinyWifi saves any re-implementation effort of the developer. Particularly, we will use TinyWifi to compare protocols developed for sensornets with standard protocols used with wireless Linux driven devices.

Kurzfassung

Wir stellen TinyWifi vor, eine neue Platform für TinyOS, dem Standard Betriebssystem für Sensornetze. Obwohl TinyOS für den Einsatz in eingebetteten Systemen entwickelt wurde, erlaubt es TinyWifi, existierende TinyOS Software für Linux basierte Geräte wie PCs oder Handhelds zu kompilieren. Unsere TinyWifi (Pseudo-)Plattform passt sich nahtlos in das existierende TinyOS Codegerüst ein und erlaubt es TinyOS-Benutzern, Programme unmittelbar auf Linux auszuführen, ohne Veränderungen an der ursprünglichen Software vornehmen zu müssen.

Der primäre Verwendungszweck von TinyWifi lautet, die Benutzung von Sensornetz-Kommunikationsprotokollen in drahtlosen vermaschten Netzwerken (wireless mesh networks) nach IEEE 802.11 Standard zu ermöglichen. Dies ist generell praktikabel, da sich Sensornetze und Wi-Fi-Netzwerke grundsätzlich ähneln. Der Aufwand zur Neuimplementierung insbesondere zwecks Evaluierung in Wi-Fi-Netzwerken wird durch die Möglichkeit, bestehende TinyOS-Anwendungen ohne Veränderungen unmittelbar unter Linux auszuführen, eliminiert. Insbesondere werden wir TinyWifi dazu nutzen, um Protokolle aus dem Bereich der Sensornetze mit in Wi-Fi-Netzen eingesetzten Standardprotokollen zu vergleichen.

Acknowledgements

Thanks to my primary supervisor Muhammad Hamad Alizai M.Sc., I had the chance to choose a really interesting thesis to work on with enthusiasm and commitment. The support he gave me was essential to find the right starting points, conquer all difficulties and to push through to the completion of this thesis. It is mainly because of him that I will be part of a reputable conference on sensor networks, which suffuses me with pride. I appreciate his open-door mentality and general kindness very much and I am looking forward to work with him beyond this thesis.

It is due to Prof. Dr.-Ing. Klaus Wehrle, our respected professor, that there is such a great research group at the RWTH University Aachen. Exciting research fields, a supportive and kind team, open-mindedness and a superb working atmosphere make the ComSys group stand out. I am thankful he supports this thesis and gave me the opportunity to write it at ComSys. It is also very much appreciated that he enables me to stick to the ComSys group as a HiWi to futher promote TinyWifi.

Beyond said people I would like to express my thanks to my second examiner, Prof. Dr.-Ing. Stefan Kowalewski, respected professor of the embedded systems group at the RWTH University Aachen, for taking the time to grade my thesis.

Because writing always requires a lot of effort and one likely fails to do it good in the first place, I am very glad for any feedback I received from friends of mine.

Finally, thank you, dear reader, for being interested and reading my thesis!

Contents

1 Introduction

In this thesis we introduce TinyWifi, a new platform for the popular open-source operating system TinyOS [7], allowing to run TinyOS applications on Linux driven host devices. TinyOS is designed for use with wireless embedded nodes. A large community uses TinyOS to develop and evaluate applications and protocols in sensornets. It features a component-based architecture to provide a highly flexible framework while minimizing code size to support severely resource constrained devices. While providing any important functionality like communication protocols, schedulers and power management, TinyOS already supports more than a dozen different hardware platforms and numerous sensor boards [12].

We developed TinyWifi to be a new software based platform, which integrates seamlessly into the unmodified TinyOS programming environment. Existing TinyOS code and functionality is preserved and reused to a reasonable extend. By compiling for the new TinyWifi platform, developers are enabled to use their software on current Linux driven devices like PCs, routers, handhelds and mobile phones. TinyWifi code aims to be portable among different Linux derivatives in order to allow for a wide range of usable target devices.

While Wi-Fi networks and sensornets distinguish from one another in their applications and in the kind of participating devices, they share a significant amount of similarity. Wireless communication within the 2.4 GHz frequency band is used in both domains. Due to physical influences and radio wave interferences, routing paths in both kind of networks are highly dynamic and links between nodes are bursty. In both domains, the logical topology is a mesh network in which each and every node can only communicate data to nodes in its radio range. Hence most algorithms and protocols used in the sensornet domain are equally applicable in Wi-Fi networks and vice versa.

Nevertheless, building and evaluating new software for a second network domain requires a disproportionate amount of additonal implementation effort. On this account, evaluation and utilization of state-of-the-art sensornet mechanisms in the akin

Wi-Fi domain is omitted in most cases. However, researchers implicitly expect their results to be applicable in the other domain as well [3, 8, 13]. That is why the major concept of TinyWifi is twofold: We want to (1) provide a powerful platform that allows for compiling TinyOS applications for Linux driven host devices while (2) eliminating the need to alter or re-implement any of the exisiting TinyOS applications. This way we extend the applicability of sensornet protocols to the inherently similar Wi-Fi domain.

Resource rich devices like PCs or routers can handle much more information at a time than deeply embedded microcontroller powered systems. We exploit the superior processing capabilities of TinyWifi host devices to advance TinyOS software wherever sensible. Certainly, we designed TinyWifi to be accurate and behave equivalent to other TinyOS platforms at the same time.

The TinyWifi code base extends the TinyOS source files tree to allow for integrating TinyWifi easily into an existing local working copy of TinyOS. Applications for Linux are compiled by issuing “`make tinywifi`”, similar to “`make <platform>`” with other platforms. The nesC intermediate compiler produces a C source code file that is compiled to an executable binary by the locally installed GNU C compiler. TinyWifi features setting a node ID via an additional command line parameter when compiling. Cross-compiling TinyWifi binaries for Linux driven devices with a different architecture than the compiling host computer is basically possible.

In line with the primary purpose of TinyWifi, the most important components are wireless networking for communication (radio *ActiveMessaging*) and timing capabilities. Having these two components available, simple but meaningful applications can be developed. TinyWifi uses Linux *sockets* and *pthreads* to communicate data in the TinyOS split-phase fashion, while all timing capabilities of TinyOS are derived from a single realtime Linux timer.

Nevertheless, TinyWifi supports all important hardware independent TinyOS functionality as well. Among those are LEDs, serial *ActiveMessaging* and sensing. We provide displaying the node’s virtual LEDs via console output. This is important since the LEDs are the obvious way to indicate any status of the node. Serial *ActiveMessaging* is primarily used by a base station application to communicate data from and to a TinyWifi application. Finally, sensing data provides a way to generate some information that is communicated among the nodes.

In the remainder of this thesis we will provide important background information in chapter 2. While chapter 3 describes the design decisions in detail, we discuss the TinyWifi implementation in chapter 4. Our evaluation results on TinyWifi are presented in chapter 5 and the thesis is concluded with chapter 6.

2

Background and Related Work

This chapter presents some useful background information on relevant subjects. It provides knowledge that probably results in a better understanding of the main part of this thesis. Sensornet devices are presented in section 2.1, including use cases and hardware design, while basic information on sensornets and Wi-Fi networks is given in section 2.2. Afterwards we discuss important insights of TinyOS to support comprehension of TinyWifi design decisions and implementation in section 2.3. Concluding this chapter is section 2.4, covering two important Linux related topics, namely sockets and realtime timers.

2.1 Wireless Sensor Network Devices

The most important difference between modern wired computer networks like ethernet or wireless networks like Wi-Fi and a sensornet is the kind of devices that participate in the network. While computers and modern mobile phones are comperatively resource rich devices, the nodes in a sensornet, also called *motes*, are small devices and provide very limited resources. That is why motes are special purpose devices that are developed for use in a specific deployment, whereas computers, mobile phones, and routers are multi-purpose devices, which are capable of handling several tasks simultaneously.

2.1.1 Use Cases

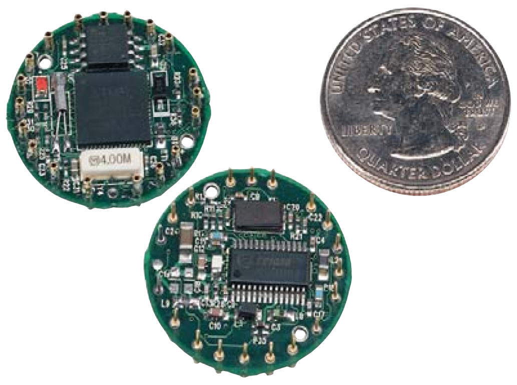
The primary field of a mote is to observe one or more physical parameters like temperature, humidity, acceleration, pressure or brightness. In more sophisticated scenarios, video or audio data can also be gathered to a limited extend. In order to make use of the raw data a mote collects, it is processed locally on the mote's hardware and communicated towards a base station. The data is then further refined by a resource rich computer and interpreted by the users afterwards. Typical application scenarios for a sensornet include:

- **Animal Habitat Observation** [18]
  Motes can be mounted on animals in order to record data within the natural environment of the animals. Tracking and sound analysis are popular use cases for biologists to learn about the subject. With the use of sophisticated communication protocols,[1] data reaches the base station even if the animals of interest live souterrain or cover a spacious habitat.
- **Precision Agriculture** [6]
  For the purpose of increasing crop yields, motes can be utilized to monitor growth and health of plants. This enables farmers to react quickly to unintended developments, thus preventing financial losses. Additionally, monitoring growth and status enables to plan harvests in greater advance and to increase the range of products.
- **Traffic Monitoring and Control** [2]
  Traffic flow and density monitoring helps preventing traffic congestions and minimizing traffic light induced delays. Intelligent sensornets can be developed to autonomously redirect traffic and fine tune traffic lights to make the traffic flow smoothly.
- **Intelligent Buildings** [17]
  The amount and location of people in a building, outdoor temperature and humidity, time of the day or properties of certain rooms are parameters for controlling air flow and temperature, humidity or lights. Sensing and acting sensornet motes enable a building's infrastructure to take care of important parameters without the need of supervising per hand.
- **Vulcano Monitoring** [19]
  Because eruptions of a vulcano are hard to predict, vulcanos are interesting research objects for geologists. Sensornets are used to monitor seismic activity without the need to provide additional infrastructure like electricity or wired network.

2.1.2 Basic Mote Hardware

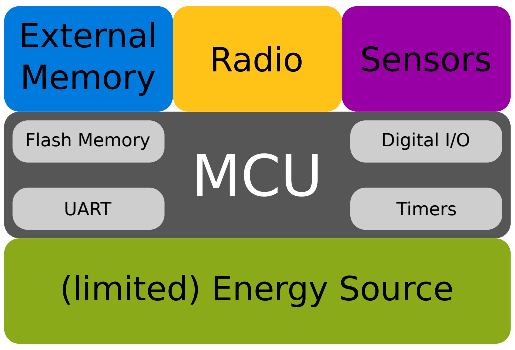
There is numerous different mote hardware already available, like the Mica2Dot mote by MEMSIC Inc. shown in figure 2.1a, which is one of the smallest popular motes. Nevertheless, for some projects new motes are developed. This is primarly due to very special demands or new integrated hardware being available. Motes vary in properties like processing power, memory size, I/O capabilities, energy consumption, radio range or size. Nevertheless, the architecture depicted in figure 2.1b is common to the great majority of motes.

[1]Routing protocols for delay tolerant networks (DTNs) allow for routing in networks without permanent reachability among nodes, most often caused by movement of the nodes.



(a) Mica2Dot Mote by MEMSIC [10]

Note that this mote is as small as a Quarter Dollar coin (25 mm in diameter). It features 128K byte of flash memory, 512K byte external memory, an integrated radio and weighs 3 g excluding batteries.



(b) Mote Architecture

The MCU (microcontroller unit), powered by an electric energy source, provides integrated features like timers and UARTs. External memory, the radio and sensors are additional integrated hardware connected to the MCU.

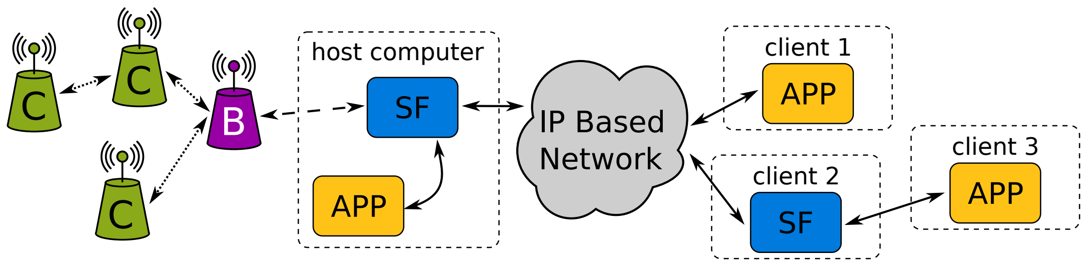
Figure 2.1 Sensor Node Hardware

The main component of many embedded system and especially of any mote is the microcontroller. It is a tiny integrated piece of hardware whose internal architecture resembles the architecture of a computer. However, on the account of size and energy consumption, particularly the processing power and memory size are not comparable to any computer in use today. Besides the main memory, the microcontroller provides non-volatile flash memory that holds space for the software the mote executes. Although the software is programmed prior to deployment and resides on the mote, sophisticated mechanisms like dynamic operating systems make reconfiguration possible.

The microcontroller is driven by a system clock an internal or external clock source provides. In order to derive timing capabilities, the system clock is used to supply counters of different widths at configurable speeds. Knowing the frequency of a counter, a fixed amount of increments (counter ticks) represents a fixed amount of time. Therefore one or more compare values are configured to create a signal "*timer expired*": If the counter value equals the compare value, an interrupt is generated and the microcontroller executes a special part of the software to service this event. Note that a microcontroller provides several counters with at least one compare register each and that counters run independently from the program execution, because extra hardware implements them.

Sensors are either attached to the mote on a modular basis or they are provided directly on the mote's circuitry board. In most cases, sensors are integrated electronic devices as well. Connected to the microcontroller through a bus, sensors provide information on relevant properties of the environment. Typically, sensor data is collected with a certain frequency either indicated by a signal from the sensor meaning data is available (interrupt) or by polling when a preconfigured timer expires.

Although there are ways to produce electrical energy on site, ranging from solar panels to energy harvesting of movements or the bloodstream, motes are basically



Serial Forwarder in Sensornets

Client motes C can communicate with one another using the radio (dotted arrows). A base station mote B serves as gateway between the wireless domain and a host computer, using a serial connection (dashed arrow). The serial forwarder *connects to the appropriate serial port and provides messages through the unified* serial forwarder packet source*, which several applications can connect to at once. Interacting with the mote is possible even remotely, e.g. via the internet.*

Figure 2.2 Functioning of the Serial Forwarder in Sensornets

powered by batteries to make them independent from any infrastructure. Certainly, this circumstance results in new challenges: Energy becomes a sparse resource as well and the lifetime of a mote consequently becomes finite.

2.1.3 Mote Communication

The radio is utilized to communicate data wirelessly to another mote in range. Data that has been successfully preprocessed by the microcontroller is transferred to the (in most cases separated) radio chip. The program is executed independently from the radio chip, and the completion of a transfer is eventually signaled by the radio chip. Because the destination mote is often not in radio range of the sender, a routing mechanism is used in the sensornet, so messages reach their destination in a hop-by-hop fashion. Since sending radio messages is one of the most energy consuming operations, wireless communication is used wisely.

Serial communication is utilized to enable interaction between host computers and motes. In conjunction with an RS232 serial converter or an USB serial converter, the UART (Universal Asynchronous Receiver Transmitter) of the microcontroller allows for a serial connection between mote and host computer. The serial line is used for programming the mote and exchanging data with the mote. Interaction with the sensornet becomes possible for a computer when using a base station: Motes running the BaseStation application convert wirelessly received data, send it to the host computer via the serial connection and vice versa.

Because a mote is connected with a single host computer at the same time, TinyOS provides a *serial forwarder*. It allows for multiple users to interact with a connected mote simultaneously, as shown in figure 2.2, and is available in three different programming languages to provide portability. Once executed and set up with the appropriate serial port, the data is converted properly and the message payload from the mote is sent to applications registered with the *serial forwarder* and vice versa.

2.2 Sensornets versus Wireless Mesh Networks

Many motes, like the Telos [16] and Mica Series by MEMSIC [9], use ZigBee compliant radio modules. These modules rely on the IEEE 802.15.4 standardization for wireless personal area networks (WPAN), whereas ZigBee provides higher level communication protocols based on the IEEE 802.15.4 specifications. The IEEE standard is intended to be used by low-speed, low-cost devices comsuming less power, so IEEE 802.15.4 suits most sensornet applications.

The standard is specified to be used within the 868 MHz frequency band (Europe), 902-928 MHz frequency range (North America) or the very popular 2.4 GHz frequency band (worldwide). The latter is also occupied by the popular standards of the IEEE 802.11 family, which is the underlying specification for wireless local area networks (WLAN). Although lower frequencies provide a higher range due to physical implications, the 2.4 GHz band is also popular for the use with motes, so WLAN capable devices and sensornets possibly operate in the same frequency band. Note that the very popular term “Wi-Fi” is a trademark of the Wi-Fi Alliance that certifies devices using WLAN and IEEE 802.11, but we use Wi-Fi as a synonym for the IEEE 802.11 standard.

Wireless communication is unreliable, because radio waves (eletromagnetic waves) are influenced by nearly any obstacle in the propagation direction. Electromagnetic waves can scatter on objects, are reflected by them and get absorbed by matter. Especially moving objects and moving radio sources/receivers are problematic, because movement causes highly dynamic link quality. Weather also has a significant influence on reachability among nodes and interference among radio waves can render information opaque. The discussed influences on wireless communication are the same for sensornets and WLANs, of course. Using the same frequency band emphasizes this similarity. Summing up, links between both motes and WLAN devices are dynamic and bursty, resulting in unreliable routing paths within the network.

Messages travelling through the network are forwarded greedily and each mote acts as an individual router that has to decide where to forward a message to (if not destined to the mote itself). Wirelessly networked devices are deployed with overlapping radio ranges, so that each device has a link to more than one other device and varying link qualities. This way communication between motes can still be performed although some motes are temporarily not available or fail. Because the links between nodes resemble a mesh in a graphic representation of this kind of networks, they are called *meshnet*.

2.3 The TinyOS Open-Source Operating System

The open-source TinyOS operating system is a programming environment and code framework that offers easy to use libraries enabling developers to rapidly implement and build applications for a variety of motes. It is designed for embedded sensor nodes, meeting requirements like small binary code size and advanced power management. Its event-driven execution model features quick responses to events and simultaneously allows for fine-grained power management.

Software is written in a stylized C language, called *nesC* [5], which is able to support the component-based and event-driven nature of TinyOS. A sophisticated *make* system is delivered with TinyOS, enabling compiling, building and programming software with a single command: "`make telosb install,123`" compiles the software in the current directory for a TelosB mote, patches ID 123 and programs it to the mote attached to the PC.

2.3.1 Driving Features of TinyOS

TinyOS is an event-driven operating system, meaning that the software solely reacts to events instead of running forever and processing something meaningful from time to time. When no event is being processed and no task is scheduled, TinyOS puts the underlying microcontroller hardware to a specific sleep mode in order to save as much energy as possible but being able to react to important (external) events. TinyOS components implement service routines for events like Boot.booted() or Timer.fired() and might *signal* another component's event.

To account for the concurrency multiple processing hardware introduces, operations on sensors, the radio or integrated functionality like the UART are executed in a *split-phase* fashion. Numerous TinyOS library calls (e.g. AMSend.send()) trigger a complex operation but return immediately afert. Instead of polling for the completion of the task, TinyOS configures the hardware to indicate such an event. Meanwhile, TinyOS services another request. Upon completion of a task, TinyOS signals the event to the application (e.g. AMSend.sendDone()).

2.3.2 Components and Interfaces

In TinyOS, components always provide a set of particular commands and react to predefined events. Interfaces determine which functionality is available in a component by describing commands and events components implementing the interface are required to provide. The basic concept behind TinyOS components is providing or using at least one interface. In order to make use of functionality provided by an interface, a developer associates his application to a component implementing the interface of interest. Command and events belonging to a certain interface are noted using a dot notation: SplitControl.start() is command start() of interface `SplitControl`.

For the purpose of minimizing the size of the final binary, functionality is only compiled in if neccessary. TinyOS applications declare components they require, whereas all but only functionality neccessary to satisfy the application's needs is compiled in automatically. Each component is either a *configuration* or a *module*. While configurations redirect interfaces they provide to other components, modules actually provide the interface by implementing the commands and events.

Configurations as well as modules consist of two parts: an *implementation* and a *declaration*. The latter defines which interfaces are used and which are provided. For configurations, the implementation part tells what other components implement

```
1 configuration ActiveMessageC {
    provides interface AMSend;
3   provides interface Packet;
  }
5 implementation {
    components LinuxActiveMessageC, LinuxPacketC, MessageHelpersC;
7   AMSend = LinuxActiveMessageC;
    Packet = LinuxPacketC;
9   LinuxPacketC.Helpers -> MessageHelpersC;
  }
```

Listing 2.1: *The configuration in this example provides interfaces* AMSend *and* Packet*. In line 6, three components are made available for use in the actual wiring. The* nesC *compiler is told in line 7 that interface* AMSend *is provided by component* LinuxActiveMessageC*. Interface* MessageHelpers *is used by component* LinuxPacketC*, and line 9 describes where to find the implementation for this interface, namely in component* MessageHelpersC*.*

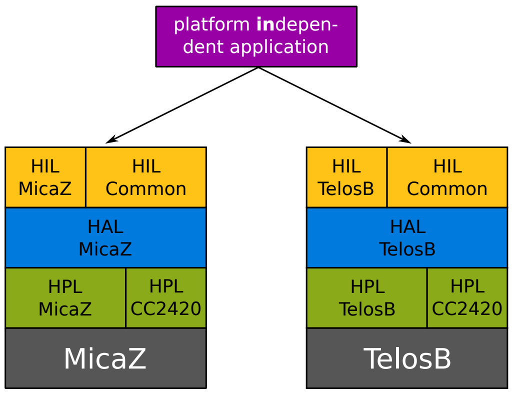
the interfaces provided. Besides wiring interfaces in their own declarations, configurations may wire foreign components as well. See listing 2.1 for a descriptive example of how components are wired together. Certainly, the implementation part in modules provides the actual software.

For the purpose of distinguishing components by their name, TinyOS specifies naming conventions for TinyOS components: Components and interfaces differ in their last letter of the file name. Interfaces carry a simple name, preferably a verb or noun, whereas names for components are terminated by a capital letter "C" or "P". While component names ending with "C" identify a configuration and thus may be used to wire an aplication with, components with "P" tag a private module that implements an interfaces provided by another configuration.

2.3.3 TinyOS Hardware Abstraction Architecture

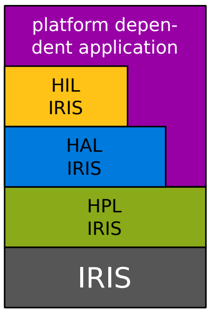
TinyOS uses a hardware abstraction architecture to support a variety of hardware platforms for the same application while maximizing code reuseability and performance, as described in TEP 2 [11]. Although abstraction typically conflicts with energy and code efficiency, it is neccessary to hide hardware subtleties and allow for portability. The TinyOS abstraction architecture, depicted in figure 2.3, consists of three layers, specifically the HPL (hardware presentation layer), the HAL (hardware adaptation layer) and the HIL (hardware interface layer). Each layer fulfills a specific purpose, which is explained in the following:

1. **Hardware Interface Layer (HIL)**
   Components belonging to this layer provide hardware independent TinyOS functions and events, such as timers. Applications use only components belonging to the HIL in order to be portable among different mote hardware. However, no mechanism is installed to check whether applications violate this rule.



(a) Platform Independent Applications

HIL components may have a platform specific implementation. Nevertheless, they provide platform independent functionality. Although very hardware specific, HPL components may be shared by different platforms if the respective motes share hardware as well (like the CC2420 radio chip).



(b) Platform Dependent Applications

When making use of functionality other than those provided by the HIL, applications become platform specific and are thus not portable in general. TinyOS does not employ a mechanism to prevent using non-HIL components, so hardware specific components may be wired to the application as well.

Figure 2.3 TinyOS Hardware Abstraction Architecture

2. **Hardware Presentation Layer (HPL)**
   Being the layer closest to the hardware, components in the HPL offer the hardware's capabilities as functions of reasonable fine-grained functional blocks. Nevertheless, HPL components are part of platform specific implementations and not intended to be used by the application.

3. **Hardware Adaptation Layer (HAL)**
   Advancing the HPL, this layer further progresses the hardware functionality to hardware independency. In contrast to the HPL, the HAL holds states and functionality that requires multiple hardware operations.

Due to the component-based nature of TinyOS, code reuse is simple when porting TinyOS to a new hardware platform. However, developing support for a new platform demands a lot of re-implementation in the HAL and especially in the HPL when using hardware not akin to already supported hardware. In order to develop a fully functional platform, the developer concentrates on the HIL functionality. *ActiveMessaging* (serial and wireless), timing, digital I/O (e.g. to drive LEDs) and sensor support are the most important parts of the HIL.

2.4 Network Communication and Timing in Linux

The ability to communicate data over a network of connected devices as well as timing capabilities are important functionalities to drive a TinyOS running node. Both of the following sections will support the reader in understanding the functioning of TinyOS on top of the Linux operating system regarding these subjects.

2.4.1 Sockets in Linux

To enable inter-machine communication within a network like the internet, a company's intranet or private computer networks, the internet protocol (IP) is widely accepted and in use. Because of several demands, the internet protocol version 4 is slowly replaced by version 6. That is e.g. to satisfy the need for new internet addresses in order to supply every participating node with an unique identifier (IP address). Due to the fact that the number of internet ready devices rapidly grows beyond the amount of addresses provided by the IP version 4 address space, unallocated identifiers are scarce.

Modern operating systems offer a simple way to make use of internet protocol based communication, although the real user space implementation to actually use IP based communication might be complex. Using sockets, programs can communicate data to remote or local applications, while the operating system handles most of the work. This includes managing protocols running on top and underneath the IP layer within the internet protocol communication stack. Data is sent to a socket and delivered eventually to the receiver while incoming data addressed to a specific socket is transparently made available to the application using it. For network nodes having more than one network interface installed, data delivered through a socket is not bound to a particular interface but is sent on those interfaces providing a connection to the destined host.

There are three basic types of IP sockets in Linux: Transmission Control Protocol (TCP) sockets, User Datagram Protocol (UDP) sockets and raw sockets. TCP and UDP sockets cover the complete internet communication stack (not considering the application itself) and only deliver the actual payload of a message. Raw sockets however leave most control to the application, especially requiring it to carry TCP/UDP and IP headers around the message. TCP and UDP separate from each other by one particular feature: TCP sockets provide connection oriented services while UDP is not using a connection but only delivering a packet independent of any other message. Note that sockets not only allow for a convinient way to exchange data between two applications on different hosts but also on the same host.

Sockets are the endpoints of inter-application communication and are unambiguously identified by the IP address and the port number. While the IP address identifies the host computer, the port number is associated with exactly one process at that particular host computer. In Linux, sockets are special files that are referenced by an integer number. Numerous system calls allow for allocating a socket, configuring and using it.

2.4.2 Timing for Linux Processes

The Linux operating system provides three different (interval) timers per process, only one of which provides realtime timing and is therefore suiteable for TinyWifi. The *itimer* is designed as both an interval timer and single shot timer. To provide both functionality even mixed up together, an *itimer* is configured with an interval value and a timer value. The latter is constantly decremented according to the process' activity and when it hits zero, a signal is generated.

Linux delivers this expiration signal of the *itimer* to the process belonging to the respective *itimer* and said process is then responsible of servicing the event. If – in the moment of expiration – the interval value is non-zero, the timer is reset to that interval value. Linux features both setting up and starting an *itimer* as well as reading the remaining value on an *itimer*, which is important to derive TinyOS timing capabilities.

To ease handling of *itimer*s, two important programming structs are used, namely *itimerval* and *timeval*. While a *timeval* struct consists of two integer numbers representing seconds and microseconds, an *itimerval* is made up of two *timeval* structs being the timer value and interval value. Summing up, the realtime *itimer* provides a sufficient resolution (microseconds) as well as satisfying accuracy on modern Linux driven systems to source TinyOS timing capabilities.

3

Design

In this particular chapter, the outline of TinyWifi is discussed. Initially, we list our design goals in section 3.1, before covering the TinyWifi architecture in section 3.2. Going deeper into the details of TinyWifi, timing capabilities are presented in section 3.3 and intricacies of TinyOS's split-phase operations are explained afterwards in section 3.4. Subsequently, principles of TinyWifi's wireless and serial communication facilities are addressed in section 3.5 and section 3.6, respectively. Finally in section 3.7, the use of sensor data in TinyWifi is explained.

3.1 Hard and Soft Requirements

The main goal of TinyWifi is enabling execution of applications written for TinyOS on Linux driven networked devices like PCs, mobile phones or routers. We want to extend the TinyOS framework to generate executables for the Linux operating system by compiling for the new TinyWifi platform. This procedure should be equivalent to compiling for any mote platform. By rewriting all important components to make them usable on Linux, developers should be able to compile applications issuing a single command "`make linux`". In order to achieve this, the *make* system already delivered with TinyOS is extended to support this operation. The result is an executable binary, which, once executed, shows equivalent behavior as the same TinyOS application running on motes.

Since our long term objective is to evaluate protocols used in the sensornet domain in inherently similar Wi-Fi networks, TinyWifi must behave similarly to other platforms. Certainly, this includes inheriting the component based nature and split-phase operation feature of TinyOS. On that account, all hardware independent functionality popular motes like the Telos provide, should be available with TinyWifi, ensuring that TinyOS applications are executable on Linux without modifications. Nevertheless, at the same time we exploit the availability of resources not offered by motes but by resource rich devices like PCs when useful. Especially for the purpose of buffering incoming and outgoing packets, this approach is sensible.

For the purpose of enabling TinyWifi to become widely accepted eventually, the following rather indirect features influence the design in general:

1. **Preserving Exisiting Structures**
   The TinyOS source tree will be utilized non-invasively, leaving it compatible to all current platforms. The TinyWifi specific code integrates seamlessly into the existing TinyOS source tree. This way, users need only one TinyOS environment to develop code for both common motes and TinyWifi.

2. **Portability Among Different Linux Versions**
   By using portable C program code, we facilitate TinyWifi to be used with different versions of Linux without the need to modify components. Since we use very common Linux system calls and mechanisms only, we envision TinyWifi to be easily portable to different Linux derivatives for routers and foreign architectures.

3. **Maximize Code Reuse**
   TinyOS already supplies a lot of code and functionality. We designed TinyWifi in a way that we can make use of existing code and do not re-implement functionality which is already available.

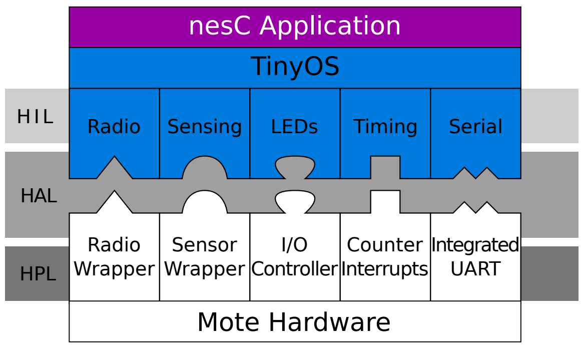
4. **Robust Software Implementation**
   TinyWifi code features robustness in unexpected situations by programming defensively and using error checking whenever sensible. Additionally, expressive error and warning messages help TinyWifi users to quickly identify code snippets causing unintended behavior.

3.2 TinyWifi Architecture

As depicted in figure 3.1a, Hardware presentation layer (HPL) components interact directly with the real hardware. Components in the hardware adaptation layer (HAL) serve as a link between HPL and hardware interface layer (HIL), by simply wiring components or – more likely – by adding advanced mechanisms to the rather simple HPL functionality. Certainly, HIL functionality is the same for every mote platform, so the right choice to make a cut is the HAL, where HIL and HPL are connected to one another.

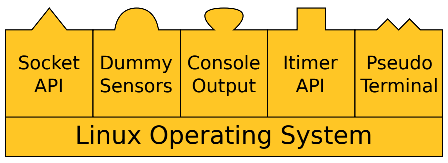
On a common mote, hardware features are presented as software functionality by the TinyOS abstraction architecture. In the case of TinyWifi, this is already done by Linux. Instead of building on top of the hardware, we use Linux to derive the TinyOS related functionality and make it available to the application.

Prior to developing, we decided what hardware technology to replace with which Linux facility. Figure 3.1b illustrates the replacement module for the bottom part of TinyOS's hardware abstraction architecture. HPL/HAL elements outlined in figure 3.1 are the most important ones and we discuss the parallelism between each corrosponding element in the following:



(a) Layered TinyOS Architecture on Motes

TinyOS features hardware independent (HIL) capabilities to use the hardware, e.g. the radio and sensors, while components within the hardware adaptation layer (HAL) provide access to the very basic implementation of hardware functionality in the hardware presentation layer (HPL).



(b) Replacement HPL/HAL

Re-implementations of HPL and HAL components built on top of Linux replace the respective components for use with mote hardware. Some HIL components may be replaced as well but still are fully compatible to other HIL components.

Figure 3.1 Layered TinyOS and TinyWifi Architecture

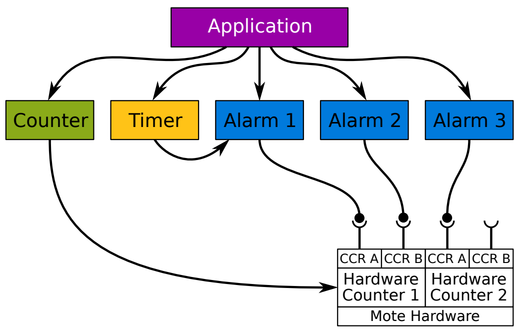
- **Wireless Communication**
  Instead of transmitting a data packet to be communicated to a radio chip, which then takes over control of the packet, we use Linux sockets for communication. These provide easy access to the network but require proper configuration of the networked node.

- **Sensing Physical Parameters**
  TinyWifi target devices are multi purpose devices and are not specifically designed for measuring data. But most Linux capable devices feature at least one real hardware sensor, e.g. to measure the processor temperature, which could be used to generate data.

- **Binary Status Indication**
  Most motes provide LEDs for status indication, driven by digital I/O pins of the microcontroller. Instead of using real LEDs on Linux driven devices – if at all available –, we generate messages to standard output to indicate the status of three virtual LEDs.

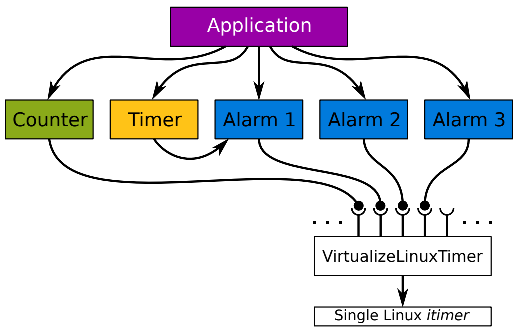
- **Counters and Timers**
  Like every digital processing machine, TinyWifi target devices provide counters and timers just like mote microcontrollers. However, these are occupied by the operating system, forcing us to use timing capabilities offered by Linux. In the case of TinyWifi, we make use of the Linux realtime *itimer*.

- **Serial Communication**
  Motes are connected to a host computer in order to be programmed or to forward data from the sensornet to a host computer and vice versa (base station). For TinyOS applications using TinyWifi, we use the Linux built-



(a) **Hardware Counters**

Multiple counter compare registers (CCR) are available on a mote, utilized to derive timing capability. Note that timers rely on an alarm rather than using the hardware.



(b) **Linux *itimer***

In order to provide multiple timing sources, the Linux realtime itimer *is first virtualized. Such timer is also sourcing the counters available in TinyWifi.*

Figure 3.2 Timing Sources for Motes and TinyWifi

in *pseudo terminal* feature to send and receive data to and from a running *serial forwarder*.

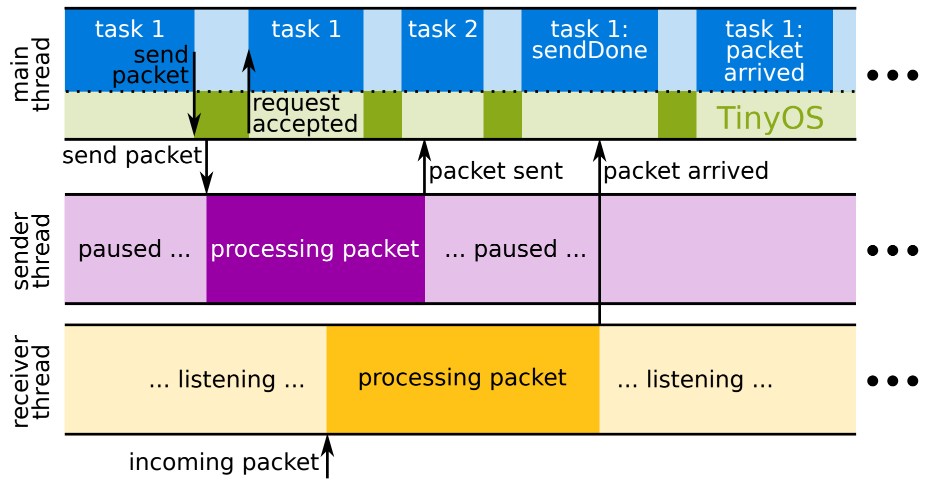
3.3 Counters, Alarms and Timers

A mote's processing facility is the microcontroller featuring counters that increment or decrement at a certain frequency (clock) in parallel to program execution. Counters are the essential method to keep track of the real time. Read accessing the counters is possible for TinyOS applications through respective components, allowing for quick and easy accessible high precision timing (for short time periods).

As shown in figure 3.2a, several compare registers for each counter allow for several simple timers to be set up. When the counter value matches one of the counter compare values, an event is triggered. TinyOS uses this simple mechanism to schedule multiple coexisting *alarms*, which basically are timers with a very low abstraction level. Besides signaling the expiration event, alarms can be scheduled, stopped and checked for their status.

In order to provide higher level timers as described in TEP 102 [11], e.g. supporting intervals, TinyOS provides components using simple alarms to derive the number of sophisticated timers needed by the application. Unfortunately, in contrast to a microcontoller offering multiple adjustable timing events, Linux provides only a single realtime *itimer* per process. This circumstance forces us to first virtualize the *itimer* functionality before being able to provide as many timers as needed by the software in the upper hardware abstraction layers.

We decided to introduce a new interface `LinuxTimer`, recreating Linux *itimer* functionality, which is used by higher level components to fulfill their orderly purpose. By virtualizing timers of type `LinuxTimer`, illustrated in figure 3.2b, we provide an abundant amount of timing sources for all timer related components. Certainly, the source to virtualize these timers is the *itimer* provided by Linux.



Exemplary Split-Phase Operation

Threads executed in parallel to the main TinyOS process allow for mimicking hardware processing in parallel. When issuing a send command, the sending thread takes over control of the message. As soon as the request has been accepted, TinyOS returns control to the application, which continues processing in parallel to the sending thread. Eventually, the thread signales the completion of sending the message to TinyOS and the respective application component. Additionally, by using a receiving thread, the main process is released from polling new incoming messages as well.

Figure 3.3 Split-Phase Operations when using TinyWifi

Since counters are no more than the representation of the time past since the counter last reached zero, TinyOS counters are simple to provide on Linux. We check the current time against the startup time of TinyOS, revealing the value of a respective hardware counter.

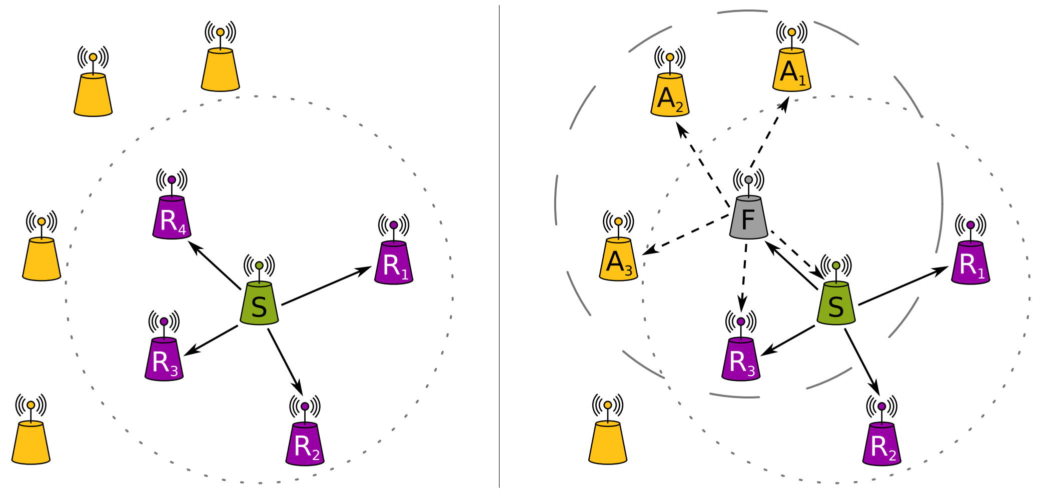
3.4 Split-Phase Operations

In section 2.3.1 we already introduced split-phase operations in TinyOS. To make said operations possible, multiple processing hardware working in parallel is required. Because we operate on top of Linux instead of controlling available hardware for ourselfs, we are unable to provide the exact same kind of split-phase operations.

The processing time for TinyOS on Linux driven nodes is comperatively short, especially on host computers with rich resources. On that account we could abandon split-phase operations and use imperative blocking instructions instead.

However, we want to provide split-phase like function calls for two reasons: On the one hand, we would also break with the requirement to behave as similar to other mote platforms as possible and on the other hand, the split-phase operations programming paradigm dictates us to return control to the caller before signaling an event indicating the completion of the respective operation.

Our primary solution to this problem is utilizing threads to imitate the parallelism of multiple processing hardware. This is reasonable since program threads are executed concurrently due to scheduling of the operating system and can be executed



(a) Intended Message Propagation

A sending node S transmits one message, which propagates omnidiretional, indicated by arrows. Nodes R_i in radio range, indicated by the dotted circle, can hear the message and process it. Note that the message might be addressed for only one of the nodes. Nevertheless, all nodes in radio range are capable of overhearing the message.

(b) Flooding in IP Networks

Assuming broadcast messages sent by S are forwarded by a receiving node F with radio range indiated by the long dashed circle. The original message by S reaches nodes A_i, indicated by dashed arrows. Since nodes A_i are not in radio range of original sender S, we need to suppress forwarding broadcast messages to achieve the sensornet behavior.

Figure 3.4 Message Propagation in Sensornets

on different processors, which provides the concurrency needed to mimic parallel processing. Instead of setting up a radio chip, we create threads, handling incoming and outgoing packets. Sending a message using the appropriate interface is delegated to the respective thread while the main TinyOS process continues working. As illustrated in figure 3.3, threads are well suited to mimic multiple processing facilities, especially on computers with multiple processors.

3.5 Radio Communication

Exchanging data is the most important task for TinyWifi, since we want to explore the applicability of sensornet communication protocols in the Wi-Fi domain. Providing this functionality becomes possible having split-phase operations and timing available. TinyOS includes `ActiveMessageC`, which is a HIL component providing high level usage of the radio. Its purpose covers enabling and disabling the radio interface, sending and receiving as well as packet manipulation ranging from timestamps and acknowledgements to attaching the payload. As any other platform, TinyWifi provides its own version of said component and wires the functionality it provides to underlying components.

When the radio is enabled by the application using the appropriate interface `SplitControl`, TinyOS is prepared to send and receive packets having created the appropriate threads. In order to send and receive data via the networked interface(s), we use Linux UDP sockets. Because we want to make the TinyWifi host device behave similar to an integrated radio on a mote, depicted in figure 3.4a, two important adjustments have to be done:

1. **Broadcasting of UDP Messages**
   Radio messages in sensornets are received by any mote in radio range of the sender and are then analysed by TinyOS. Depending on the message being destined for a particular mote, the message is passed to listening components. Nevertheless, it is also possible to sniff packets, meaning receiving messages not addressed for the particular mote.
   Usually, UDP packets are used for unicast communication. Messages not addressed for the receiving node are discarded in the lower levels of the IP stack by the operating system, hence do not reach the application. By broadcasting the UDP packets containing TinyOS messages, we ensure that packets are passed through to the application and not discarded on a lower level of the IP communication stack, since broadcast messages are addressed to all nodes by definiton. When a message arrives, it is analysed by the receiving thread and the appropriate event is triggered.

2. **Suppress Message Forwarding**
   Since we use an already fully functional communication protocol to carry messages, namely IP, we have to take care that TinyOS packets are not forwarded by receiving nodes. This is important to prevent the network of being flooded, which would result in every participating node in the network to receive the message, as illustrated in figure 3.4b.
   Instead only those nodes in range of the radio should receive the message. Although forwarding of packets destined to the broadcast address is decided by the (individual) routing layer implementation used, we ensure the correct behavior by setting the time to live (TTL) header value of the packet to zero.

As arranged by TinyOS (TEP 111 [11]), we specify the contents of the standard TinyWifi message separately for each header, footer and metadata. We have no use for data in the footer but provide information on the message in the header and use the matadata field to store timestamps and acknowledgements. The payload region is filled by the application with content of interest, using our packet manipulation mechanisms to ensure that the message is not corrupted by the application.

3.6 Serial Communication

In most scenarios, a base station mote is utilized to forward messages from within the sensornet to resource rich computers for further processing and analysing. A common TinyOS application named BaseStation is used to forward incoming messages through the serial port of the mote to another device. The host computer connected to a base station executes an instance of the *serial forwarder*, redirecting incoming mote serial messages to all applications registered with the *serial forwarder* using IP based communication.

In contrast to sensornets, resource rich Linux driven devices can easily connect directly to the network. Using a Linux capable networked node running BaseStation and an additional computer attached via a serial connection is inconvenient. Instead, we make it possible for one single node to be both base station and *serial forwarder*. This is reasonable since Linux capable devices within a meshnet offer sufficient resources to power a *serial forwarder* besides other applications. To achieve this feature, presented in detail in TEP 113 [11], we discuss two approaches:

1. ***Serial Forwarder* Packet Source**
   Serial forwarders can be daisy chained by connecting to one another through *serial forwarder* packet sources. By re-implementing the serial communication stack of TinyOS, we could transform the data and provide it to a *serial forwarder packet source*. The *serial fowarder* will then be connected to the source provided by the TinyOS process.

2. **Software Implemented Serial Port**
   A serial port the *serial forwarder* can connect to could be implemented in software. For that approach to success, the virtial port has to be available as a Linux device and behave exactly like a serial I/O port. Of course, this device should be accessible through standard Linux device drivers.

Both approaches require a lot of code and for the virtual serial port, advanced mechanisms will probably be neccessary. However, Linux provides a feature that already offers the functionality needed to substantiate the second approach. Linux *pseudo terminal*s have been part of Linux for a long time and provide two paired serial pseudo devices (ports). Using simple system calls, data written in one of the paired devices appears as readable on the other device and vice versa.

When the serial communication stack is enabled by the TinyOS application, the *pseudo terminal* is set up and TinyOS hooks itself into one of the paired ports. The name of the opposite device is disclosed to the user and the *serial forwarder* can be connected to that particular port. Note that the Java based *serial forwarder* is not capable of using the TinyWifi *pseudo terminal* serial port, whereas both the C and C++ implementations can indeed connect to it. We assume that the Java based *serial forwarder* cannot handle *the pseudo terminal* port because of Java limitations.

3.7 Sensors

We developed TinyWifi to enable researchers to evaluate sensornet protocols in the inherently similar Wi-Fi wireless domain. On that account, support for sensing data is a secondary issue. Nevertheless, some data should be available to drive data exchange within the network, since protocol evaluation is not possible without communication. By providing dummy data sources implementing interfaces used with real sensors, we allow for creating data to be sent through the network.

Most devices discussed to run TinyWifi utilize sensors, e.g. to monitor processor temperature. In some cases, these are avilable through Linux. Nevertheless, reading values from such sensors is highly platform specific and thus using real sensors to provide data on some device types is a long term aim. Instead, we use a pseudo data source like a sine generator to acquire data.

4

Implementation

Having discussed the design of TinyWifi, we now present interesting implementation related details. First, the integration into the TinyOS source tree is presented in section 4.1. A look at the interaction with standard console output is given in section 4.2. In section 4.3, the implementation of TinyOS counters and timers using the Linux *itimer* is described. Afterwards, details of our approach to provide split-phase operations are explained in section 4.4. Subtleties of TinyOS communication capabilities when using TinyWifi are covered in section 4.5 for wireless data exchange and section 4.6 for serial communication. Section 4.7 concludes this chapter, addressing the simple dummy sensor data acquisition implementation in TinyWifi.

4.1 Build System

The TinyOS source tree is reasonably well organized. The root directory holds three important folders: `apps`, `support` and `tos`. While mote applications for using and testing TinyOS can be found within `apps`, folder `support` holds tools like the *serial forwarder* and files for use with the popular Linux build tool *make* to build TinyOS applications. The last important folder within the TinyOS root directory is `tos`, which is – among others – home of TinyOS wide interfaces in subfolder `interfaces`, chip specific components in subfolder `chips` and all platform specific files in subfolder `platforms`.

Usually, new platforms are derived from the *Null* platform, which is a dummy platform implementing the crucial components as stubs to make TinyOS compile applications for it. Certainly, these applications will not work before implementing functioning components. We also started with the *Null* platform and incrementally added new functionality. The correctness of any component was tested extensively during the development to ensure the functioning prior to progressing to another component.

We hooked TinyWifi into the TinyOS build system the same way any other platform is integrated. For the purpose of preparing a new platform for the already provided *make* environment, two files are of major importance: To make TinyOS recognize the new platform, a target file must be available in folder `support/make`.[2] The file contains simple commands to declare the platform's name, adds some compiler flags and finally includes rules from a platform specific subfolder to build an application for the target. Also part of the build process is `.platform` in the platform root directory[3]. It defines additonal compiler options and folders included in the compiler's search domain for *nesC* source files. For TinyWifi, some additional compiler options are neccessary, e.g. option "`-lpthread`" is given to enable thread support.

The command line to compile an application for TinyWifi is similar to compiling for other platforms. By issuing "`make tinywifi`", the application is compiled and an executable binary file containing the program is produced. Since other platforms use "`make <platform> install,<nodeid>`" to patch the node's ActiveMessage address and transfer the binary to the mote, we also feature that syntax to define a non-default *ActiveMessaging* address. The node's ID given on the command line is used to declare a global constant via the "`-D`" option of the compiler. Our TinyWifi code then uses this global constant's value as the *ActiveMessaging* address. Additionally we offer compiling and executing with a single command line by typing "`make tinywifi install,<nodeid> run`".

4.2 TinyOS Outputs and Local Messages

For several reasons like debugging, we want to use standard output to the console to print messages on the screen. In TinyOS, there already is a printf() facility with limited platform support, redirecting messages to the serial port. On an attached host computer, the messages can then be read and displayed by a simple viewer application, also part of the TinyOS environment. Instead of redirecting messages to the serial line, we want to print messages to standard output of the console without detours by using the Linux built-in printf() command. We can achieve this by including the appropriate Linux header file, which provides printf() using standard output. Nevertheless, additonal changes to the application are required to achieve the intended behavior.

Applications using the printf() function declare the path of the printf() library in their `Makefile` in order to enable printf(). Since we want to integrate TinyWifi into TinyOS without changes to the original source code, we have to build around the exisiting printf() library. Indeed, applications using printf() can be compiled for TinyWifi, but support is limited until TinyOS is enabled to use the Linux printf() implementation. For this purpose, the path to the TinyOS printf() library must not be provided in the application's `Makefile`.

[2]Folder and file locations are given relative to the TinyOS source root, if not stated otherwise.

[3]The term "platform root directory" refers to folder `tos/platforms/tinywifi`, holding the platform specific files for TinyWifi.

```
LinuxActiveMessageC.nc: In function 'LinuxActiveMessageC$receiverThread':
LinuxActiveMessageC.nc:319: ERROR: setsockopt(...) failed!
LinuxActiveMessageC.nc:319: Linux says it was (errno): Socket operation on non-socket
LinuxActiveMessageC.nc:319: gai_strerror says it was: Bad value for ai_flags
LinuxActiveMessageC.nc:319: program exits with status -4.
```

Listing 4.1: *TinyWifi features information rich and descriptive error and warning messages. Note that line 4 is special for networking related errors and warnings.*

4.2.1 Virtual LEDs

Although Linux offers a much simpler way to provide messages to the user via printf(), LEDs are substantial for TinyWifi to support applications out of the box, since LEDs are often used on real motes. Subfolder `leds` in the platform root directory holds two files responsible for providing virtual LEDs: Configuration `PlatformLedsC` provides three interfaces `GeneralIO` as *Led0* through *Led2* and uses interface `Init`. While interface `Init` is passed through to component `PlatformC` and is used at boot time, the `GeneralIO` interfaces are wired to module `PlatformLedsP` implementing the LED functionality.

Features of LEDs defined by interface `Leds` are turning on and off as well as toggeling single LEDs. Additionally, reading and setting the LED configuration as a bitmask is possible. As on any other platform, applications wire to the HIL component `LedsC` for the purpose of accessing the LEDs. The high level features in interface `Leds` are mapped to the low level functions provided with interface `GeneralIO`.

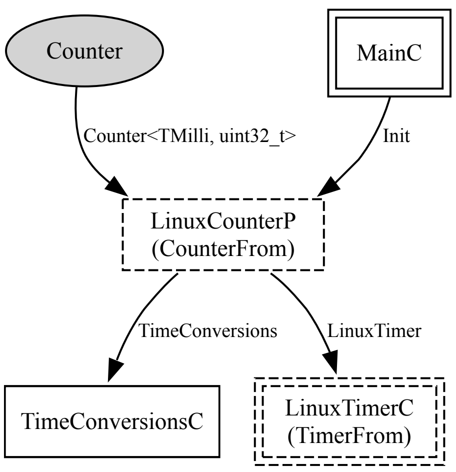
In `PlatformLedsP`, we use general helper functions to control the pseudo LED states, which are stored in a simple global array. This way duplicate code is avoided and increasing the amount of LEDs becomes simple. Moreover, the output can be configured to be either one message per LED access or one line revealing the LED's states, which is updated for every change. By defining __LEDS_SHUT_UP,[4] the output to the console can be completely suppressed.

4.2.2 Error Handling

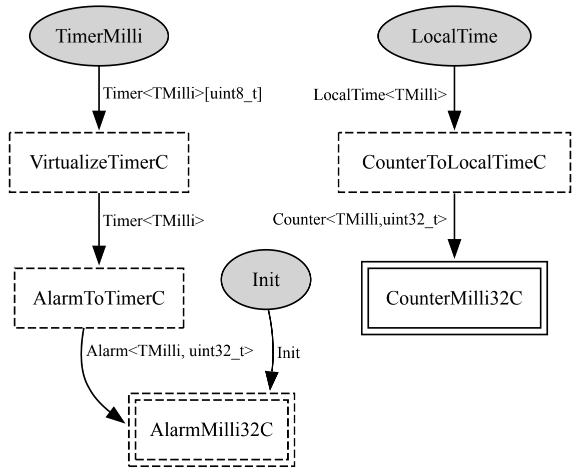
TinyWifi code is developed to be resilient for the purpose of functioning correctly in exceptional situations. The most significant part in achieving robustness is thorough error checking. Linux system calls are checked against success or failure and the appropriate error or warning message is displayed in the console.

In order to unify error messages and to offer a centrally adaptable and information rich output, see listing 4.1, TinyWifi provides `TinyWifiError.h` implementing two customized error functions. Function tinyWifiNetError() is used to output error messages when using Linux sockets, providing even more information than tinyWifiError(). The latter is a special case of tinyWifiNetError() and thus uses tinyWifiNetError() to print its messages, setting an error code of *SUCCESS* for the network related error code parameter.

[4]Constants influencing TinyOS applications compiled for TinyWifi can be found in header file `hardware.h` located in the platform root directory.



(a) CounterMilli32C



(b) HilTimerMilliC

The component graph illustrates that configuration `CounterMilli32C` provides interface `Counter` with precision TMilli *(binary milliseconds) and width of 32 bit. For the purpose of initializing when booting, it wires interface `Init` (provided) of component `LinuxCounterP` to `MainC`. To fully satisfy `LinuxCounterP`'s needs, configuration `CounterMilli32C` also wires `TimeConversionsC` and `LinuxTimerC`.*

Provided interface `Init` is passed through to `AlarmMilli32C`. Standard TinyOS component `CounterToLocalTimeC` provides interface `LocalTime` with precision TMilli *and is wired to `CounterMilli32C`, from which the local time is derived. Additionally, interface* Timer *with precision* TMilli *is provided multiple times (indicated by array notation) as* TimerMilli. *Note that the virtualized timers are derived from `AlarmMilli32C` using TinyOS component `AlarmToTimerC`.*

Figure 4.1 Example Wirings of Timing Components

The unified error messages include information about where the error occured (file and function as well as code line), a customized message (string) and the human readable description of the numerical error code, obtained by Linux calls perror() for any error and gai_strerror() for network related problems.

4.3 Counters, Alarms and Timers

TinyOS offers three different timing facilities: simple counters, single shot alarms and convenient (interval) timers. On common motes, interface `Counter` is implemented by directly reading the counter value and generating an event for counter overflows. Multiple Alarms (highly limited in number) are derived from counters in conjunction with the compare registers of a counter to generate event fired(), whereas timers (available in high numbers by virtualization) are derived from alarms. Since all timing facilities can be used in multiple resolutions and widths (of the respective value), TinyOS offers components to convert between different types, like `TransformCounterC` and `TransformAlarmC`.

In TinyWifi, counters and alarms are implemented as a generic interface each. While `LinuxCounterP` provides counters of all widths and precisions, component `LinuxAlarmP` implements alarms of all flavours. Certainly, said components are private and TinyOS applications may use them by wiring to the appropriate wrapper, like `CounterMilli32C`, `CounterMilli16C` ... `Alarm32khz32C` and `Alarm32khz16C`. Component `Counter<precision><width>C` instantiates a new `LinuxTimerC` and a new `LinuxCounterP` with the appropritate precision and width, as illustrated in figure 4.1a for a tangible example. For each component `Alarm<precision><width>C`, another `LinuxTimerC` is required, as well as a new `LinuxAlarmP` with the appropriate parameters. Every component providing an alarm or counter is given a name following the naming conventions in TinyOS to provide compatibility with existing applications.

Since the *itimer* functionality is used by several components, we take two steps in order to supply multiple components with a separate timer:

1. **Define New Timer Interface**
   For the purpose of successfully providing functionality similar to an *itimer* to several components, we introduce a TinyOS interface named `LinuxTimer`, describing the functionality needed to derive all TinyOS timing capabilities. Actually, this new interface is fairly similar to the high level interface `Timer` of TinyOS. Nevertheless, the data types change from simple integers to structs of type *timeval*, and some additonal commands are required, e.g. getStartupTime(), which returns the Linux time when the TinyOS application booted (important for counters).

2. **Virtualize Linux Timer**
   Similar to the original TinyOS component `VirtualizeTimerC`, we provide 256 instances of interface `LinuxTimer` by virtualization. The source timer for virtualization is component `NativeLinuxTimerC`, which implements said interface using the unique Linux realtime *itimer*. All other instances of components using interface `LinuxTimer` are set to wire themselves to `LinuxTimerC`, which allocates a new timer of type `LinuxTimer` through virtualization component `VirtualizeLinuxTimerC`.

Counter values are represented as simple unsigned integers and interpreted as ticks of the respective clock source. Since Linux uses seconds and microseconds to represent time, we need to convert from ticks to struct *timeval* and vice versa in many scenarios to provide the correct representation of time. We introduce interface `TimeConversions`, implemented by component `TimeConversionsC`, which provides conversion from one representation into another for any precision.

For embedded systems, the ability to wait for a specific amount of time is very important. Many external controllers are triggered by sophisticated protocols and require crucial timing. On that account, TinyOS provides interface `BusyWait`, which allows for waiting a certain small amount of time with high precision due to the blocking nature of `BusyWait`. We use Linux system function usleep(), which blocks the calling process for an amount of microseconds. The given value for command BusyWait.wait() is converted to microseconds and results in the argument for usleep().

Additionally, any platform is required to provide its version of component `HilTimerMilliC`, which is used by the frequently used HIL component `TimerMilliC`. See figure 4.1b for the wiring graph of component `HilTimerMilliC`. Interface `LocalTime` offers reading the amount of time passed since startup of the mote in a 32 bit integer, sufficient to represent approximately 8 years of runtime in milliseconds. In contrast to other platforms, TinyWifi also provides component `HilTimer32khzC`, which provides timers with 32 kHz precision as well as the respective interface `LocalTime` with 32 kHz precision.

4.4 Split-Phase Operations

Despite commands to send out messages, there are more split-phase operations in TinyOS, e.g. ReadNow.read() in component `DemoSensorP`, which is our dummy sensor implementation. Since sensing is a subsidiary feature, we do not use threads to provide dummy sensor data, saving sophisticated programming code. Instead, a timer is set up to fire in the near future by command ReadNow.read(). The respective ReadNow.readDone() event is signaled on timer expiration.

Another method to emulate split-phase operations is utilized in component `SineSensorP`, being a part of the originial TinyOS source code. A task is posted to the scheduler and executed eventually, calculating some data and presenting it along with the ReadNow.readDone() event. For both the timer-based approach and the task-based approach, control is immediately returned to the calling function and the respective "*done*" event is signaled eventually, which conforms to the desired behavior of split-phase operations.

4.4.1 Split-Phase Operations Using Threads

For the purpose of providing well-engineered split-phase operations on top of Linux, we utilize program threads. In the current version of TinyWifi, we encounter two major split-phase operations that are handled by threads: AMSend.send() and UartStream.send(), which are commands to send data via the radio and via the serial connection, respectively. Since radio chips as well as the internal UART have to be set up correctly prior to utilization, interfaces `SplitControl` and `StdControl` are used to enable and disable the radio and UART.

Although very similar in functionality, these interfaces distinguish from one another in their behavior: While `SplitControl` controls a hardware component and signals the respective "*done*" event in the future, the commands in `StdControl` are blocking functions and only return after an attempt to change the hardware state has been accomplished. We integrated the task-based approach discussed with `SineSensorP` to support split-phase operation behavior with interface `SplitControl` for component `LinuxActiveMessageC`.

In the following, we exemplarily discuss the split-phase operations in `LinuxActiveMessageC`. On call of SplitControl.start(), we create thread *receiverThread*, which first sets up the outgoing sockets and then creates another new thread, namely *senderThread*. Afterwards, the incoming socket is configured properly using Linux system calls and on success, the task reponsible for signaling SplitControl.startDone() is posted to the scheduler. At this stage, there are the main TinyOS process, a thread handling incoming packets and another thread handling outgoing packets. Since all concurrent threads alternate in execution, receiving and sending packets as well as processing the main application is done simultaneously.[5]

Until reception of a message, *receiverThread* remains in system call recvfrom(), not consuming CPU time since recvfrom() is blocking. The message is handled by the thread in order to displace all processing intense procedures out of the TinyOS thread. Afterwards, the reception of a message has to be signaled to the application component using the radio. Calling the appropriate event from within the thread would result in the thread being blocked until the application is done processing the event. On that account, we post a special task packetReceivedTask() to the main process. During further processing of incoming packets, said task is scheduled for execution and signals the arrival of the packet to the application eventually.

Thread *senderThread* is responsible for transmitting messages to other nodes in the network. Like *receiverThread*, it is kept alive by an infinite loop in which the thread tests for packets to be sent out. Since infinite polling for events consumes processor power, we developed the thread to pause after each transmission and wait for another packet to be transmitted. Again, the completion of sending a message is signaled by a task, which is enqueued by the thread and executed eventually in the near future.

4.4.2 Impact of Threads on TinyOS Scheduler

Since enqueuing tasks to signal events is used by multiple concurrently executed threads, we need to make sure that posting a task does not lead to corruption of the task queue. For that purpose, we use a mutex[6] to ensure exlusive use of function pushTask() in component `SchedulerBasicP`.

Because TinyOS is a very energy efficient operating system, the scheduler puts the microcontroller of a mote to sleep when no tasks are being scheduled for execution. Interface `McuSleep` defines commands to put the processor to sleep or update the power state. The particular component implementing this interface allows for setting the correct sleep mode per platform, depending on the sleep modes supported. Important for continued functioning is reacting to events like timer expiration or completion of a measurement. Until such an event is triggered, the MCU can power down in order to save energy.

For TinyWifi, we issue Linux system call pause(), which halts the calling process until a signal is received. This approach makes TinyOS applications running with TinyWifi on Linux driven devices behave as expected – until threads are considered: The main process yielding the processor and waiting for a signal results in not recognizing posting of a task when done by another thread. This is due to the thread being able to successfully post the task while the main process including the scheduler is halted. Hence, the newly posted task is not executed when enqueued but when the next signal is handled. To prevent this frequently occuring problem,

[5]Note that real concurrency is only achieved on systems with multiple processors, but scheduling on a single processor already allows for comparable concurrency.

[6]Mutexes allow for developing program code sections that can only be executed by one concurrent process at a time.

the function posting a task is altered to send a signal to the main process after enqueuing the task. Certainly, the signal makes TinyOS return from `pause()` and enables TinyOS to recognize the newly enqueued task immediately.

4.5 Radio Communication

For the purpose of supporting radio communication, we basically have to provide implementation for component `ActiveMessageC`, as specified by TEP 116[11]. As with most other platforms, a seperate version of this component is needed for TinyWifi. `ActiveMessageC` provides several interfaces for use with radio messaging, of which most are passed through to `LinuxActiveMessageC`:

- **Initialization Prior to Enabling**
  Interface `Init` is provided to execute code when booting. Component `PlatformP` calls command `Init.init()`, implemented in `LinuxActiveMessageC`, which sets up mutexes for the use with threads.

- **Controlling the Radio**
  As discussed in section 4.4.1, interface `SplitControl` is used to control the radio state, i.e. to enable or disable it. We use this interface to set up the message handling threads (enabling radio) or to destroy them (disabling radio). Since the threads are part of the core messaging functionality, module `LinuxActiveMessageC` provides said interface.

- **Sending Messages**
  The main command for sending messages is included in interface `AMSend`, which also features accessing the actual payload and the maximum payload length of a message. Event `AMSend.sendDone()` indicates the completion of a previous call to `AMSend.send()` in a split-phase fashion. Certainly, `LinuxActiveMessageC` is the right module to implement interface `AMSend`.

- **Receiving Messages**
  Using `ActiveMessageC` allows for two ways of receiving messages: It is possible for an application to (1) sniff all overheard packets or (2) receive only messages addressed to the node, which is the usual behavior. An application wires itself to either `ActiveMessageC.Snoop` or `ActiveMessageC.Receive`, respectively. `ActiveMessageC` provides both interfaces through `LinuxActiveMessageC`.

- **Packet Manipulation**
  To allow for portability and protecting the internal message structures around the payload, TinyOS requires applications to manipulate packets using predefined interfaces like `AMPacket`. Properties like source address, source group, and message type can be accessed via `AMPacket`, although this is also done by the *ActiveMessaging* communication stack automatically. Module `LinuxPacket` provides both interfaces `Packet` and `AMPacket`.

- **Packet Acknowledgements**
  It is possible to request an acknowledgement when sending packets, by using

interface `PacketAcknowledgements`. It allows for setting and resetting the request flag as well as checking whether the message was acknowledged by the recipient. We implemented some of this functionality in `LinuxPacketAcksC`, whereas `LinuxActiveMessageC` is responsible for recognizing and sending the actual acknowledgements. TinyOS specifies that the sendDone() event for outgoing messages requiring acknowledgement is only signaled when the status of the particular message is fixed. This rule implies that it has to be possible to check sent messages for their acknowledgement status in the sendDone() event handler. For that reason, a timout timer is started when sending messages with the acknowledgement flag set. On timeout expiration, the message is consequently considered not acknowledged.

4.5.1 Communication Setup

Prior to using the radio messaging facility, threads, sockets, buffers, and variables have to be set up correctly. In SplitControl.start(), appropriate action is triggered depending on the current status. If the messaging facility is actually not ready for sending, we create thread *receiverThread*. As required by a split-phase operation, the function call returns immediately after thread creation. The actual setup is done in parallel by said thread.

Two outgoing sockets are inquired by *receiverThread*: One of these is prepared for *senderThread*, the other will be used by *receiverThread* to send acknowledgements. After issuing socket() twice to get two new socket file descriptors, the sockets are configured to broadcast messages, i.e. to use both the internet protocol and MAC broadcast address. Additionally, the time-to-live header field is manipulated to be TINYWIFI_TTL (value 0 by default) for any outgoing message by setting the appropriate socket property. In the next step, *senderThread* is created, which is now enabled to start sending packets.

A third socket file descriptor is created in a similar way as the outgoing sockets. The receiving thread calls bind() in order to gain exclusive access on the specific communication end point. Prior to entering the infinite loop and starting to receive messages, we catch the internet protocol addresses of the host machine and save them into a local data structure to sort out looped back packets later. Finally, startDoneTask() is posted to the scheduler, eventually signaling the SplitControl.startDone() event while *receiverThread* enters an infinite loop and waits for incoming packets.

Since we do not want the sending thread to poll for new outgoing packets, we utilize pause(), but also need a signal to wake it up from pause(). Invocation of a signal handler is neccessary, as pause() will not return on reception of unhandled signals. On that account, *senderThread* registers signal handler sendPacket() first. Afterwards, *senderThread* enters an infinite loop and halts until the wake up signal is received.

Certainly, disabling the ActiveMessage communication stack is also possible. Function SplitControl.stop() is intended to cancel all ongoing operations and free resources. In TinyWifi, we set a cancellation point for each thread, close the sockets and free some memory. Again, a task is posted to complete the second phase of the split-phase operation, namely signaling SplitControl.stopDone() to the application.

4.5.2 Receiving Radio Messages

Returning from the blocking Linux system call recvfrom() to receive data out of the datagram socket, *receiverThread* first tests if the radio interface has been shut down via SplitControl.stop() in the meanwhile. Issuing pthread_testcancel() results in termination of the thread if a cancellation point was set by SplitControl.stop() beforehand. Otherwise, the current message is examined and handled properly.

Since broadcast messages are also sent back to the local host, we have to look for looped back packets. By comparing the sender's address of each UDP message with the list of local internet protocol addresses, we can determine whether the packet is of interest.

The possibility to request and send acknowledgements asks for several extra steps: For incoming acknowledgements, we flag the acknowledged local message, cancel the acknowledgment timeout timer, post sendDoneTask(), and wait for the next incoming data. If any incoming TinyOS message demands for an acknowledgement, a simple acknowledgment packet with empty payload is built and sent to the originator by *receiverThread* immediately.

On motes, the radio can handle only one message at a time. When data arrives, it is collected by the respective event program code of the application. If a second message arrives prior to fetching the data, one of the messages is lost. Since TinyOS applications compiled for the new TinyWifi platform are executed on resource rich Linux driven host devices, we provide a ring buffer for incoming messages. Its size (number of messages) can be influenced by adjusting the constant TINIWIFI_RECVBUF in header file `platform_message.h`, which also holds TINYWIFI_TTL and message struct definitions along with related parameters.

The timestamps for incoming packets represent the time of arrival, so timestamps are set for non-looped messages at this stage. Afterwards, the message is stored in the receive buffer. For the purpose of protecting the ring buffer against corruption, we use a blocking mutex to grant exclusive access to it. The mutex affects packetReceivedTask() and *receiverThread*. Finally, right before *receiverThread* loops and listens for another message, packetReceivedTask() is posted, eventually signaling reception of the message to the appropriate application components.

4.5.3 Sending Radio Messages

If the radio facility has not been setup correctly, a call to AMsend.send() returns the appropriate error code. Otherwise, the current message is prepared to be sent over the radio. An exeption has to be made if *senderThread* is still busy sending another message, which is properly handled by returning error code *EBUSY* if the mutex fails to lock due to the sending thread being busy.

Because AMSend.send() is a split-phase operation, most processing should take place from within *senderThread*. Hence we only fill the send buffer and trigger *senderThread* by sending the preconfigured signal, which will cause the thread to return from pause(). Control is quickly restored to the caller and all processing intense operations are done by the separate sending thread.

The message is prepared to be sent via the socket in *senderThread*. Operations include resetting the acknowledgement flag, setting the source and destination address, as well as setting other important header fields using the `Packet` and `AMPacket` interfaces. The timestamps are set directly after the message has been sent to achieve accuracy. A notice is put out to the console if the Linux command sendto() failed for some reason. The error code passed to AMSend.sendDone() later is adjusted properly after completion of the operation.

Subsequently, either sendDoneTask() is posted or the acknowledgement timeout timer is started. In the latter case, AMSend.sendDone() is either posted on expiration of the timer or on reception of the appropriate acknowledgement message. For evaluation and performance purposes, the acknowledgement timeout can be easily adjusted in header file `platform_message.h` with constant TINIWIFI_ACK_TIMEOUT.

4.6 Serial Communication

Component `PlatformSerialC` in the platform root directory suffices in order to implement the serial messaging facility. That is because we can cut very low in the hardware abstraction architecture, due to the fact that the serial communication is byte oriented, hence there is few difference in using an UART on a microcontoller and writing/reading to/from a Linux file descriptor. The sophisticated techniques used to provide serial messaging are very similar to those in `LinuxActiveMessageC`, since both feature split-phase sending and receiving data operations.

Interfaces provided by `PlatformSerialC` are less in number than in component `LinuxActiveMessageC`: Interface `StdControl`, which is the non-split-phase alternative of `SplitControl`, is used to enable and disable the serial communication facility. Mutexes are also required and therefor initialized by command init() of interface `Init`. The actual serial messaging functionality is defined by interfaces `UartStream` and `UartByte`.

In contrast to `LinuxActiveMessageC`, the setup is done completely in StdControl.start(), since this function is blocking and reports the success or failure of starting the communication stack on return. Neccessary operations include opening the *pseudo terminal*, saving the file descriptors of both serial ports provided by openpty(), granting access to them and creating one sending and one receiving thread. Finally, the device name of the software serial port and its speed are printed out on the console to ease indentification of said port. The respective stop command closes all file descriptors and cancels both *senderThread* and *receiverThread*.

Thread *receiverThread* reads data available byte-by-byte and processes them individually. If the interrupt for incoming bytes is enabled, each byte is signaled to UartStream.receivedByte() by a separate task. A ring buffer is used to deposit incoming bytes and to make them available to the signaling task. Additionally, the current byte is written to another receive buffer provided by the application, if applicable. Function UartStream.receive() provides this buffer and configures an amount of bytes to be received (counter). For each byte written to the buffer, *receiverThread* decrements the counter. When the buffer is finally full, receiveDoneTask() is posted, signaling UartStream.receiveDone() in the near future.

Command receive() of interface UartByte requires special treatment. It features a blocking read of a single byte from the serial input with a timeout of several characters. Either one byte is available and its value is given to the caller or the failure of the command is indicated on timeout expiration. In order to provide this functionality along with the receiving thread, we decided for another use of pause(). Function UartByte.receive() registers a signal handler, sets the timeout timer and then calls pause(). On reception of a byte, *receiverThread* makes the newly received byte accessible and wakes up UartByte.receive(), which then delivers the single byte with status *SUCCESS*. Otherwise, the timeout timer expires, the event handler flags the operation as failed, and wakes up UartByte.receive() returning status *FAIL* to the caller.

4.7 Sensors

We provide component DemoSensorC, implementing both interfaces Read and ReadNow through DemoSensorP. Since sensors are always queried via said interfaces, existing applications using sensors can be compiled for TinyWifi requiring only marginal modifications: By re-wiring to DemoSensorC, all applications' needs for sensors can be satisfied when using TinyWifi. In contrast to Read, interface ReadNow uses commands and events preceded with keyword *async*, which tags functions save to execute within interrupt handlers.

We provide different data sources for the two interfaces. For Read, we use TinyOS component SineSensorC as the sensor source. Since SineSensorC already features the split-phase operation Read.read(), we simply exploit SineSensorC to provide the Read interface. Applications using interface ReadNow in TinyWifi's DemoSensorC will sense a triangular signal, derived from the linearly rising local time. To substantiate the split-phase operation ReadNow.read(), we use another timer that fires in the near future. On expiration of that particular timer, ReadNow.readDone() is signaled along with the pseudo-sensor value.

5

Evaluation

For the purpose of evaluating the functionality of our solution, we extensively tested the features of TinyWifi. First, in section 5.1, we discuss that building TinyOS applications for Linux is possible when using TinyWifi. Subsequently, we present different test applications examining the timing functionality in section 5.2, before we have a look on the serial communication in section 5.3. Finally, we briefly discuss the wireless communication functionality for both simple neighborhood communication and multihop communication in section 5.4 and section 5.5, respectively.

We primarily tested TinyWifi on our development notebook, which is a 2.0 GHz dual core machine with 3 GByte memory running Ubuntu 10.04. Test applications by TinyOS as well as newly developed simple and advanced tests were used to examine the functioning our implementation. If not stated otherwise, we used a virtual machine, also running Ubuntu 10.04, to provide another node for communication when neccessary.

5.1 Proof of Concept and Portability

Before testing our TinyWifi platform in detail, we try to build simple TinyOS applications with TinyWifi to run on Linux and see if the result is useful.

Besides the *Null* platform, there is also a `Null` application, which uses interface `Boot` and implements event `Boot.booted()`. Since the only event it implements is designed to do nothing, its only purpose is to show if compiling for our new platform works. In the case of TinyWifi, the result of entering command "`make linux`" is a C source file `build/linux/app.c`, generated by the *nesC* compiler and a binary executable, subsequently produced by the locally installed *gcc* using the `build/linux/app.c` source file.

Compiling the `Null` application – besides many other applications – is successful. The make process generates an executable `build/tinywifi/main.exe` out of the `build/linux/app.c` source file derived by the *nesC* compiler. The executable binary can be directly executed on Linux driven devices.

A similarly simple application is the TinyOS PowerUp application, which enables the first LED after booting. On execution of PowerUp after successful compiling, we indeed notice that the LED status line appears and *LED0* is turned on as expected. It shows that or solution to provide LED support works. Applications Blink and Oscilloscope, which also use LEDs, emphasize the correctness.

While our primary Linux derivative on which we developed and tested TinyWifi is Ubuntu 10.04, we also successfully repeated the build process on Fedora Core 13 and openSuse 11.3, representing the three major branches Debian, Slackware Linux and RedHat, respectively. Being able to run applications on different Linux versions, we show that our TinyWifi code is portable to a variety of Linux derivatives. In any case, we found at least application Blink and Oscilloscope working correctly.

5.2 Timing and Sensing

In order to test the timing functionalities developed for TinyWifi, we wrote several applications by ourselfs to check for the correct behavior of alarms, counters and timers:

- **LinuxTimerTest and VirtualizeLinuxTimerTest**
  In order to evaluate the functioning of the `LinuxTimer` interface implementation using the Linux *itimer*, namely in component `NativeLinuxTimerC`, we developed application LinuxTimerTest. Upon completion of the boot procedure, we check whether the startup time is set correctly. We configure the timer to fire once in 1 s relative to 250 ms in the past, which is by design possible and tested successfully at this stage. On timer expiration, getLeftover() and stop() are evaluated for correct functioning, prior to setting up the timer as an interval timer. We await timer expiration 5 times and check whether the result is accurate. Additionally, VirtualizeLinuxTimerTest allows for testing the correctness and accuracy of multiple `LinuxTimer` instances provided by our virtualization facility. We find the results to be fully satisfiying in regard to accuracy and funtionality.

- **LinuxAlarmTest**
  TinyWifi alarms already rely on the working underlying Linux *itimer* virtualization. In this test case, we use 4 alarms of different widths and precisions. The alarms are each set up to fire at different times in the future. The primary objective of this test is to evaluae the accuracy of alarms. Additionally, we test command isRunning() and stop() on one of the alarms, which is representative for all other, since they rely on the same generic implementation (`LinuxAlarmP`). Again, the result is fully satisfying regarding accuracy and funtionality. Note that this test also shows that the `LinuxTimer` interface virtualization works as expected.

- **LinuxCounterTest**
  This particular test application is designed to print the difference between the expected counter values and the counter values retrieved through the respective interface. We use all 4 available counter flavours and continuously compare the expected and retrieved values with each other. We observe the overflow signaling for correctness as well. During development, this test revealed a lot of errors in the first implementation of counters but now shows that the counters are usable as expected and provide high accuracy.

- **BlinkAdvanced**
  This application uses all flavors of interface `BusyWait` and calls wait() on all of them with different values. Prior to calling BusyWait.wait(), we save the current uptime and check the uptime after the return of BusyWait.wait() for plausibility. Subsequently, we check the high level timing capability of TinyOS by allocating two timers of precision *TMilli* and one of precision *T32kHz*. On each timer expiration, the uptime in the respective precision is printed out. The tests show the correct functioning of interface `BusyWait` and prove that multiple high level timers can be used at the same time, while providing satisfying accuracy.

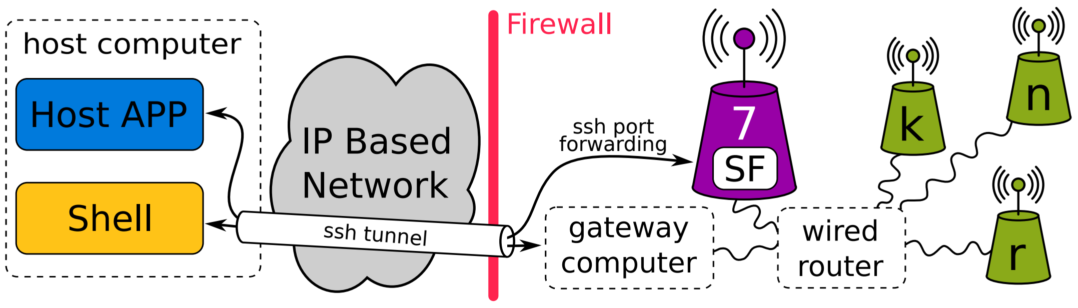
TinyOS offers a simple test application Sense, which reads the `DemoSensorC` sensor module and displays the bottom three bits of the reading with help of the node's LEDs. By compiling and running Sense, we show that our dummy sensor implementation as well as the LED output work properly. Note that application MultihopOscilloscope, presented in section 5.5, that the data from the sensor is reasonable.

5.3 Serial Communication

While evaluating the serial communication stack implementation of TinyWifi, we encountered difficulties. Executing the same binary on different machines resulted in different bahavior: In contrast to the development system, the serial line worked only infrequently on the testbed nodes (outgoing direction). We were unable to identify the cause of this behavior in the first place, but due to the indeterministic occurence of this failure, we suspected a timing and concurrency issue.

Indeed, we had to ensure that the sending thread is instantly ready to transmit data when signaling StdControl.start(). Additionally, we had to move the processing of the outgoing data from the thread function to the signal handler that was first intended to only wake up the thread function from pause().

TinyOS provides a test application for the serial port named TestSerial. It consists of a TinyOS application and a Java based host counterpart, which both send incrementing counter values to each other. The host application prints the counter value on the screen, while the node displays the three least significant bits on its LEDs. This test application works exactly as expected. Additionally, in use by the TinyOS BaseStation application and MultihopOscilloscope, the virtual serial connection is also found to work correctly in either direction.



Testbed Evaluation Setup

We use a secure tunnel to gain access to the gateway computer and setup a port forwarding bypassing the firewall to node number 7, the selected root node. Despite the actual application, node 7 also runs a serial forwarder*. The data from the* serial forwarder *is tunneled back to the host computer's evaluation application. We control the testbed nodes (k, n, r, etc.) by using* ssh *on the gateway computer to execute commands remotely. Although the testbed nodes are additionally connected with each other by Ethernet, the test application uses the wireless interface by setting the appropriate broadcast address.*

Figure 5.1 Evaluation in the UMIC Meshnet Testbed

5.4 Wireless Communication

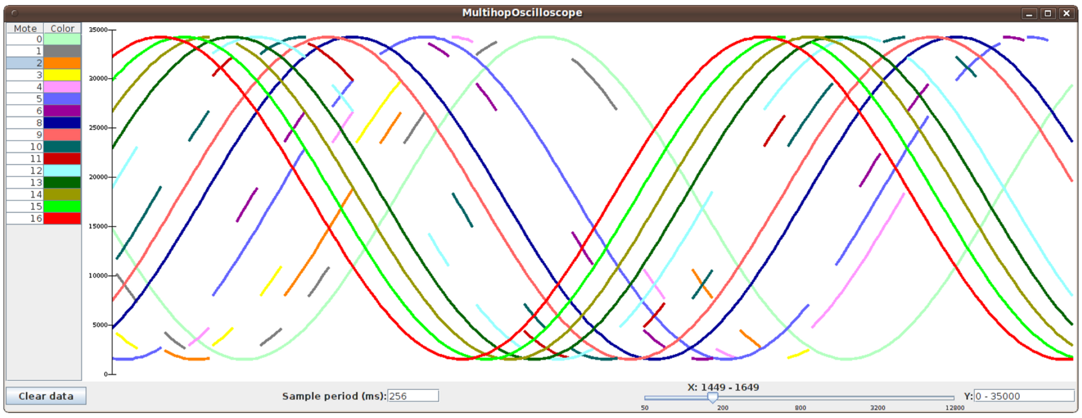
Although we use mechanisms for enabling and disabling the wireless communication stack that are very similar to the methods used with the serial communication, we did not encounter similar concurrency problems. Nevertheless, the implementation was revised several times during the development, thanks to early evaluation: We learned, that we have to filter out looped back packets that arrive through the standard Linux loop back interface. The coordination of the threads providing split-phase operations revealed itself to be sophisticated and it was during the early evaluation, that we found out having to wake up the main thread on posting tasks, as discussed in section 4.4.2 earlier.

The evaluation of wireless communication reveals the correct functioning by starting applications using the wireless communication stack, like Oscilloscope or RadioSense-ToLeds, which can successfully communicate data in the intended manner.

For the advanced mechanisms of the wireless communication facility, we created test applications. Running these newly developed test applications, we can show that acknowledgements are treated correctly. Beyond that, the timestamps of packets are accurate and set like specified by the appropriate interface. The reception of a packet and the completion of sending a packet is signalled promptly.

5.5 Multihop Communication

In order to show that TinyWifi can fulfill its primary purpose, we configured 16 nodes in the UMIC wireless mesh network testbed to run MultihopOscilloscope, which uses an actual sensornet routing protocol to collect data, namely the collection tree protocol (CTP).



***MultihopOscilloscope* Visualization**

The screenshot of the MultihopOscilloscope *Java based host application shows the data from the dummy sensors of 16 different nodes in the UMIC wireless mesh network testbed. Only few nodes are in radio range of the collecting node (logical address 0), which indicates that the collection tree protocol (CTP) of TinyOS works properly with TinyWifi on Linux driven wireless networked nodes.*
Due to packet losses, curves can be discontinuous. The nodes are synchronized properly, so that the curves do not overlap. We slightly changed the implementation of the Java based host application: We (1) altered the background color of the plot area to be white and (2) set the stroke width to a higher value. With these modifications, the readability of the output is increased tremendously, especially for the printed version.

Figure 5.2 TinyOS's MultihopOscilloscope Host Application Output

The collection tree protocol is part of TinyOS and provides a best-effort anycast connectionless communication service. It features forwarding messages to preselected root node(s). CTP uses trees to determine the next hop forwarding a message greedily towards the root node(s). The protocol is explained in much more detail by TEP 123 [11].

We configured node 7 (physical numbering) to run the MultihopOscilloscope with wireless address 0 (logical numbering). This makes node 7 the root node collecting all the data and forwarding it over the virtual serial line.

On the very same node, we ran a *serial forwarder* that connects to the virtual serial port of the root node and advertises the messages to a specific TCP port. We established an IP tunnel for that specific port from our host computer to node number 7 over *ssh* in order to receive the data from the *serial forwarder*. We can then visualize the data that arrived at the CTP root node. The setup used is illustrated in figure 5.1 and also shows the configuration we will use when evaluating other sensornet protocols in the UMIC meshnet testbed.

We successfully received data from the sine sensor of all 16 of the configured nodes, as shown in figure 5.2. Hence, we finally reliably showed that we can use TinyWifi to evaluate sensornet protocols designed for use with TinyOS in meshnets.

6

Conclusion

In this thesis we presented TinyWifi, a new TinyOS platform enabling convenient and robust platform support for Linux driven host devices. Due to the inherent similarity of sensornets and meshnets, communication protocols for TinyOS can now easily be evaluated in Wi-Fi networks using TinyWifi. Because TinyWifi integrates smoothly into the current TinyOS source and supports all important hardware independent functionality, existing TinyOS applications can be compiled for Linux driven devices without modifications. This approach saves a lot of re-implementation and allows researchers to evaluate their sensornet protocols in the Wi-Fi domain, instead of just (implicitly) claiming the applicability.

During the development of TinyWifi, we faced architecture and design related problems of TinyOS, like split-phase operations, that demanded sophisticated solutions in order to provide similar paradigms on top of Linux. We found a way to derive powerful timing functionalities from only a single Linux realtime timer and constructed a fully functional radio messaging facility working equally well in comparison with motes. We also found a good way to provide a serial port running the TinyOS application by using Linux pseudo terminals.

Finally, we showed that TinyWifi is functioning as expected in a variety of different situations. We found that our approach is suitable to evaluate sensornet protocols in meshnets with less time effort.

6.1 Future Work

Although TinyWifi already supports a range of different applications and is functioning correctly, some tasks are left to be dealt with in the future to further refine TinyWifi code and finally use it for research.

Linux Raw Sockets

Currently, we use UDP/IP packets to transmit TinyOS messages to other nodes. At the same time, we prune the main purpose of the internet protocol layer, namely routing, by setting the “time to live” header field to a value forcing the receiving node to not route the packet to foreign nodes.

Nevertheless, this approach works well and saves a lot of sophisticated code. Additionally, this method could be implemented and tested rather quickly, as required for this thesis. But by sending messages through the internet protocol communication stack of Linux without using its features, we cause latency as well as processing and data overhead. Although this overhead is comparably negligible on resource rich Linux driven devices, we want to use Linux raw sockets to get even closer to the behavior of motes and cut back the overhead. Additionally, we do not rely on a properly configured routing table, since data using raw sockets is sent out on the interface regardless of the nerwork configuration. The effort to replace the socket type so is expected to be limited to a single component and requires testing the messaging interface again.

Wireless Extensions

In some sensornet applications (e.g. link estimators), additional parameters like the received signal strength indicator (RSSI) are of interest. Currently, we do not provide support for such information, since obtaining them is hardware specific and limits portability. Nevertheless, we will implement such functionality for use in our meshnet testbed and get even closer to the subtle features of motes.

Sencosrnet Protocol Evaluation

This thesis aimed at providing the basic infrastructure to allow for using existing TinyOS applications on Linux driven devices without the need to make great changes or even re-implement them each individually. The next step after progressing to raw sockets and using wireless extensions is starting to evaluate sensornet protocols in the Wi-Fi domain.

In the near future, we intend to compare sensornet protocols to existing protocols currently used with Wi-Fi networks in a scientific way and to present our findings to the research community.

Contribute TinyWifi to TinyOS Repository

TinyWifi is primarily targeted to extend applicability of state-of-the-art sensornet communication protocols to Wi-Fi networks. Hence, the TinyWifi platform support should be available to researchers for allowing evaluation of sensornet protocols in meshnets in the future. TinyWifi integrates seamlessly into the existing TinyOS source code, allowing to compile applications for motes as usual while having Linux platform support at the same time.

On that account, we intend to contribute TinyWifi to the official TinyOS repository as soon as TinyWifi proved itself to be reliable in use for our own evaluations.

Bibliography

[1] Alizai, M. H., Landsiedel, O., Bitsch Link, J. A., Goetz, S., and Wehrle, K. Bursty Traffic over Bursty Links. In *SenSys'09* (November 2009).

[2] Chen, L., Chen, Z., and Tu, S. A Realtime Dynamic Traffic Control System Based on Wireless Sensor Network. In *ICPPW'05: Proceedings of the 2005 International Conference on Parallel Processing Workshops* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 258–264.

[3] Fonseca, R., Gnawali, O., Jamieson, K., and Levis, P. Four-Bit Wireless Link Estimation. In *HotNets* (November 2007).

[4] Fonseca, R., Ratnasamy, S., Zhao, J., Ee, C. T., Culler, D., Shenker, S., and Stoica, I. Beacon Vector Routing: Scalable Point-to-Point Routing in Wireless Sensornets. In *NSDI* (April 2005).

[5] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN PLDI* (June 2003).

[6] Langendoen, K., Baggio, A., and Visser, O. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (April 2006), p. 8 pp.

[7] Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., and Culler, D. The emergence of networking abstractions and techniques in TinyOS. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (March 2004).

[8] Mao, Y., Wang, F., Qiu, L., Lam, S. S., and Smith, J. M. S4: Small State and Small Stretch Routing Protocol for Large Wireless Sensor Networks. In *NSDI* (April 2007).

[9] MEMSIC Inc. Mica2, Mica2Dot, MicaZ, Telos, IRIS, Imote2 and Cricket Mote Datasheets available online in pdf format. `http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html`.

[10] MEMSIC Inc. Mica2Dot Datasheet, Document Part Number: 6020-0043-06 Rev A. `http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html`.

[11] TinyOS Online Documentation. TinyOS Extension Proposals (TEPs). `http://www.tinyos.net/tinyos-2.x/doc/`.

[12] TinyOS Online Homepage. Mission Statement. `http://www.tinyos.net/special/mission`.

[13] Newsome, J., and Song, D. GEM: Graph EMbedding for routing and data-centric storage in sensor networks without geographic information. In *SenSys'03* (November 2003).

[14] Perkins, C. E., Royer, E. M., and Das, S. R. Ad hoc On-Demand Distance Vector (AODV) Routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications* (February 1999).

[15] Polastre, J., Hill, J., and Culler, D. Versatile low power media access for wireless sensor networks. In *SenSys'04* (November 2004).

[16] Polastre, J., Szewczyk, R., and Culler, D. Telos: Enabling Ultra-Low Power Wireless Research. In *IPSN* (April 2005).

[17] Singhvi, V., Krause, A., Guestrin, C., Garrett, Jr., J. H., and Matthews, H. S. Intelligent light control using sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), ACM, pp. 218–229.

[18] Szewczyk, R., Polastre, J., Mainwaring, A. M., and Culler, D. E. Lessons from a Sensor Network Expedition. In *EWSN* (2004), pp. 307–322.

[19] Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M., and Lees, J. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing 10* (2006), 18–25.

List of Figures

A

TinyWifi Source Code Compact Disc