University of Latvia
Faculty of Computing

Atis Elsts

A Framework to Facilitate Wireless Sensor Network Application Development

Doctoral Thesis

Field: Computer Science
Subfield: Data Processing Systems and Computer Networking
Scientific Advisor: Prof., Dr. Sc. Comp. Leo Selavo

Riga – 2013

Abstract

Wireless sensor networks is a technology that was originally envisioned as a tool for broad range of purposes and target audiences. However, wireless sensor network software development is complicated, because it brings together the complexity of resource-constrained embedded system software and the complexity of distributed system software. This is one of the reasons why sensor networks have not yet become an ubiquitous technology.

This problem is partially solvable by decreasing the accidental software development complexity that is at the moment present in the process of sensor network application development. The sensor network application developer must be provided with appropriate development tools, languages and libraries in order to allow him to focus on the essential complexity of software development. Therefore this dissertation presents a number of approaches that facilitate sensor network application programming, including: a new application development language that allows to write applications in more concise and less complex way, and an approach to increase the resource-efficiency of sensor network operating systems. The approaches are integrated in a unified framework, along with other software designed by the author and his research group.

This framework is targeted towards two categories of users: firstly, novice programmers, secondly, professional programmers, including programmers with limited experience in wireless sensor networks.

The evaluation shows that these approaches lead to applications with decreased resource usage and lower code complexity.

Keywords: Wireless sensor networks, domain-specific languages, operating systems

Acknowledgments

First of all, I would like to thank my supervisor Leo Selavo for introducing sensor networks to me, and for his helpful suggestions during my doctoral studies and thesis-related research.

Second, since the time I started research for my dissertation, I have formed the opinion that working and achieving results in the systems subfield of computer science is very much a collaborative experience. This thesis, as it stands now, would not be possible without my coauthors: Rihards Balass, Janis Judvaitis, Artis Mednis, Girts Strazdins, Reinholds Zviedris, and other colleagues at the Institute of Electronics and Computer Science (EDI).

The staff at Latvia State Institute of Fruit-Growing (Edgars Rubauskis, Ieva Kalnina, and others) receives thanks for the time they devoted to evaluation of our wireless sensor network software, and for their help in field experiments.

Administrative personnel at the University of Latvia (both at Faculty of Computer Science and at Faculty of Physics and Mathematics) and Institute of Electronics and Computer science receives thanks for their helpful and kind attitude when confronted with questions or requests for organizational help.

Many thanks to Inga, including for her help in field experiments and useful suggestions, but mostly for her support in general.

European Union receives thanks for supplying the funding. Thesis-related research was supported by the European Social Fund, project No. 2009/0219/1DP/-1.1.1.2.0/09/APIA/VIAA/020, and the European Regional Development Fund, project No. 010/0317/2DP/2.1.1.1.0/10/APIA/VIAA/142. The finalization of this thesis was also directly supported by the European Social Fund, project No. 2011/0054/1DP/1.1.2.1.2/11/IPIA/VIAA/002.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog-to-Digital Converter, a microcontroller that converts from continous quantities to discrete (numeric) quantities
API	Application Programming Interface, a source-code based interface between software components
BPML	Business Process Modeling Language, a language for business process modeling
COTS	Commercially Available Off-the-Shelf, hardware that is sold in commercial markets and in large quantities
CPU	Central Processing Unit, the core unit that executes instructions in a computer
CSMA	Carrier Sensor Multiple Access, a channel sampling based channel access method for shared medium networks
DSL	Domain-Specific Language, a type of programming language limited to a particular domain
EDI	Institute of Electronics and Computer Science
GCC	GNU Compiler Collection
GNSS	Global Navigation Satellite System, sattelite-based positioning system
GPS	Global Positioning System, a widely used example of GNSS
IDE	Integrated Development Enviroment, a software application that serves as a tool for sofware development, and commonly provides editing, building and debugging tools
IP	Internet Protocol
LED	Light-Emitting Diode, a kind of semiconductor light source
LQI	Link Quality Index, a numerical measure of radio link quality
MAC	Media Access Control, a sublayer in the data-link layer in networking stack architecture that provides channel access mechanisms
MansOS	Multi-platform Ad-hoc Netted Operating System
MCU	Microcontroller Unit, a small integrated circuit that contains processor, memory, and input/output ports

OS	Operating System, a set of software that serves as an intermediator betwen user applications and computer hardware, and provides common services to the user
PDR	Packet Delivery Ratio, the proportion of received versus sent packets
POSIX	Portable Operating System Interface, a family of standards for operating systems
QoS	Quality of Service, mechanisms and policies related to speficic requirements for data transfer in computer networks
RAM	Random Access Memory, a form of data storage that allows data to be accessed in arbitrary order. Typically used for volatile data storage
RSSI	Received Signal Strength Indication, a numerical measure of the power received via radio link
RTC	Real Time Clock, a microchip that keeps track of the current time
SAD	"Sensori Augļu Dārzā", informal name of our environmental monitoring project
SEAL	**Se**nsor Network **A**pplication Development **L**anguage
SI	International System of Units (Le Système international d'unités), a commonly used system of units of measurement
SPI	Serial Peripheral Interface Bus, a digital data transmission architecture and protocol that is characterized by serial, three-wire data transfer
SSMP	Simple Sensor Management Protocol, a management protocol used in MansOS shell interface
TDMA	Time Division Multiple Access, a schedule-based channel access method for shared medium networks
USART	Universal Serial Asynchronous Receiver/Transmitter, a hardware implementation of serial data communication interface
WSN	Wireless Sensor Network, a system that consists of spatially distributed autonomous sensor nodes

Introduction

Among the components that are needed to make computing truly pervasive, wireless sensor network technology is one of the key pieces in the puzzle. Small, spatially distributed and interconnected autonomous nodes with low energy requirements and low price can truly change the way computers are perceived and used. They can enable the people to interact with the environment and with each other in ways previously unimaginable. They can allow us to capture the important information that is out there, in the physical world, but inaccessible and unobserved at the present. Or can they?

Programming wireless sensor networks is difficult enough to hinder the fulfillment of their promise. Along with other factors, this issue during the last decade has turned WSN from a new and exciting research area into a much-researched technology that is still on a brink of unlocking its true potential. The sensor network business is expected to become "multibillion dollar industry", but only when there is "major progress with technology" [BusinessWire 2012].

Wireless sensor network programming is still an unresolved research problem [Welsh 2010]. Their software is inherently complex because it brings together the complexity of resource-constrained embedded system software and the complexity of distributed system software. Still, a distinction between *essential* and *accidental* complexity can be made [Brooks 1987]. Essential complexity stems from the the inherent nontrivial structure of software artifacts, while accidental complexity is caused by the approaches used to create and manage those artifacts. Progress in software engineering can only decrease the accidental complexity. We[1] believe that at least wireless sensor network *application* (rather than system) programming complexity at the moment is essential only partially. It is partially accidental because of the mismatch between the skills & experience of the sensor network application programmers (both current and potential ones) from one side, and the software libraries, tools and interfaces for sensor network programming from the other side. As a consequence, the accidental complexity can be decreased by providing these application developers with better fitting software libraries, tools, and interfaces.

For instance, the existing sensor network operating systems are characterized either by a steep learning curve for system programmers without sensor network experience, or by suboptimal use of resources. This thesis shows that these drawbacks can be ameliorated.

Furthermore, sensor network applications at the moment are usually developed by the computer science professional. This approach leads to several difficulties for the sensor network end user, who has limited programming skills. First, it requires participation of a qualified computer scientist in the sensor network development and deployment processes. Second, it requires constant availability of the profes-

[1]Throughout the thesis this term is used to denote the author individually, rather than the members of his research group.

sional to perform maintenance operations. As a consequence, the labor costs of qualified engineers adds up to the largest cost factor in real-world sensor network deployments [Corke 2012]. The problem is exacerbated by the fact that many networks are deployed in remote areas that are far more frequently visited by domain experts than by computer scientists. Finally, it puts the domain expert in the role of a passive user rather than an active contributor. Therefore, there are both financial and scientific incentives to empower these users and allow them to write and maintain sensor network applications on their own. One may speculate that these issues will became even more important in the future, if expenditures for hardware will go down in proportion to expedintures for software creation and maintenance, as it has already happened [Van Vliet 2008] in the areas of desktop and mainframe computing.

A number of sensor network programming approaches at medium or high level of abstraction already has been proposed by the research community. Some of the approaches are targeted specifically towards domain experts. However, compared to them, the results of this thesis:

- are integrated in a single framework, therefore their usability is improved by offering a number of interfaces in different levels of abstraction, suitable for distinct groups of users, and resource usage efficiency is improved by offering cross-layer optimizations;
- are suitable for a wide range of sensor network applications, instead of being restricted to solving a particular programming problem or to a particular type of application;
- take into account research results about novice programming;
- have been experimentally tested in preliminary user studies and in a real-world use case.

Assumptions

This work is based on the assumptions (discussed in detail in Chapter 1 and supported by research literature) that:

1. A significant part of the prospective wireless sensor network application developers are so-called *novice programmers*, who are unskilled in any of general purpose programming languages.

2. Excluding this novice programming group, the remaining application programmers are more likely to have C programming language experience than specific sensor network operating system programming language experience. They are also likely to prefer to implement at least some of common sensor network applications using a thread-based, rather than event based model of concurrent execution.

3. The programming approaches (languages, libraries, and middleware) suitable for wireless sensor network application development should be made more accessible for these two target audiences.

Theses

The following theses are put forward in this work:

1. The accessibility of sensor network application programming approaches for novice programmers can be increased by offering these programmers:
   - a domain specific language with high level of abstraction;
   - an integrated development environment;
   - a visual programming interface.
2. From application programmer's point-of-view, several improvements are possible in the design of wireless sensor network operating systems compared to the state-of-art. They include:
   - improving resource usage efficiency for typical applications, by offering a semiautomatic, modular component selection mechanism;
   - improving learnability (for already experienced programmers) of sensor network operating systems without loss of resource usage efficiency, by providing the application programmers with programming models and software abstractions that are appropriate for wireless sensor networks, and are either generally known, or are well-known among system programmers.

Research objective

The objective of this thesis is to develop a prototype of resource-efficient software framework that would facilitate the development of wireless sensor network applications for two groups of users:

1. novice programmers;
2. professional programmers, including programmers with limited experience in wireless sensor networks.

Brief description of methods

The tasks performed during the development of this thesis includes conducting a literature survey. The outcome of the survey includes description and comparison of 23 wireless sensor network programming approaches and 4 sensor network operating systems.

For the experimental part of the thesis, a number of software prototypes has been developed and analyzed. The analysis includes quantitative measurements using software engineering metrics (cyclomatic complexity, lines of code, programmer's effort according to Halstead complexity measures) and other metrics (binary code size, RAM usage, upload time, energy consumption). Usability tests of the software were conducted, with 30 participants in total. Finally, practical evaluation of the functionality of software on real sensor network hardware has been performed for several months in outdoor conditions.

Author's contribution

Most of the approaches and the software described in the thesis are developed personally by the author. For example, the SEAL programming language was created and implemented by himself. Important exceptions are:

- MansOS sensor network operating system was initially envisioned by Leo Selavo, and later developed by himself and a number of people. More detailed list of contributions is available at the Introduction section of Chapter 2.
- SEAL integrated development environment was developed by Janis Judvaitis under author's supervision.
- SEAL-Blockly language and development environment was developed by Janis Judvaitis and Leo Selavo, with only a small author's participation.

SADmote sensor device hardware was designed by Rihards Balass and Leo Selavo with author's participation.

Thesis outline

The thesis includes introduction chapter, four body chapters, conclusion chapter, and one appendix.

Chapter 1 gives a brief overview of wireless sensor network hardware and its specific properties. After this, the chapter overviews the target audience of this work, the specifics of it, and the software that is appropriate for this audience. The rest of the chapter is devoted to the results of the literature survey.

Chapter 2 describes system level support for wireless sensor network application development framework, specifically, the MansOS operating system. The chapter in detail describes several aspects of MansOS (code architecture, component selection mechanism, concurrent execution models, etc.) that are directly relevant to this thesis. The chapter also includes results of MansOS evaluation and describes two of its applications.

Chapter 3 includes one of the cornerstones of this work: the description and evaluation of the wireless sensor network application development language SEAL. The design choices of SEAL are described and justified. The place of SEAL among the related work is briefly discussed, as well as its elements, architecture, and the software development tools built on top of SEAL. Substantial qualitative and quantitative evaluation results of four aspects of SEAL are given. The evaluation includes complexity evaluation of the language and applications written in it, efficiency evaluation, applicability scope evaluation, and usability test results.

Chapter 4 describes a wireless sensor network application for environmental monitoring in precision agriculture. The system itself already has been described in more detail in author's publications; dissertation focuses on what lessons were learned from this sensor network use case.

Chapter 5 (the conclusion) lists the main results and conclusions from this work, as well as author's publications and presentations related to the dissemination of research results.

The appendix includes the grammar of SEAL.

Chapter 1

Programming wireless sensor networks

Contents

1.1 Introduction

Despite the relatively small size, sensor network software is quite complex [Picco 2010] and its development is characterized as "difficult" [Welsh 2004] [Casati 2012] [Chu 2007] and "challenging" [Rubio 2007]. Indeed, the complexity of sensor network software comes from two aspects: it includes the specific complexity of *embedded* software and the specific complexity of *networked* software. WSN challenges already familiar to the developers of networked software include need for distributed algorithms, unreliable communication, the need for QoS guarantees, and the need for synchronization. WSN challenges already familiar to embedded system developers include hard computational resource constraints, nontrivial debugging, and (for a subset of WSN software) the need for real-time operation. Furthermore, sensor networks place new, specific constraints on the software, most prominently, the *energy constraint*. Since sensor networks often are powered from low-capacity and/or non-rechargeable power sources, persistently continuous operation is not an option; energy-efficient software with a low duty cycle should be used. All of this shows that specific software should be developed for sensor networks; reusing existing software (such as operating systems or network protocol implementations) is not the best solution in most of cases.

The designer of an application development framework has to work at different levels of *abstraction* (in contrast, a user may choose to work only at one level). The following quote shows how this term ("abstraction") is defined in the existing research literature: "An abstraction for a software artifact is a succinct description that suppresses the details that are unimportant to a software developer and emphasizes the information that is important. For example, the abstraction provided by a high-level programming language allows a programmer to construct algorithms without having to worry about the details of hardware register allocation. Software typically consists of several layers of abstraction built on top of raw hardware." [Krueger 1992].

More specifically to this dissertation, two critical levels of abstraction are separated: the *system* level and the *high-level language* level. Our contributions at the system level of abstraction are described in Chapter 2; our contributions at the higher level of abstraction are described in Chapter 3. In contrast, the current chapter reviews the state of art in sensor network programming at both levels: first, at the system level (Section 1.4), then at the high level (Section 1.5).

The current chapter starts with a brief introduction into sensor network hardware (Section 1.2). After that, it turns its attention to sensor network users that are novice programmers (Section 1.3). (In the scope of this thesis, a *novice programmer* is defined as a person not skilled in any of general purpose programming languages.) We believe this group of users is an important stakeholder in the future of sensor networks. The rest of the chapter is devoted to survey of state-of-art sensor network software (operating systems and programming approaches).

1.2 The hardware of sensor networks

All software is ultimately constrained by the hardware on which is executing. Hardware constraints are especially important in the context of embedded systems.

It is typical of the existing wireless sensor deployments [Strazdins 2013] to use specific hardware devices: *motes*, either custom or commercial off-the-shelf. Tmote Sky [Moteiv Corporation 2006] (Fig. 1.1) is a prominent example. It is also an example of a *TelosB* hardware architecture compatible mote.

A mote typically consists of several integrated blocks:

- a low-power microcontroller;
- a radio transceiver;
- on-board sensors;

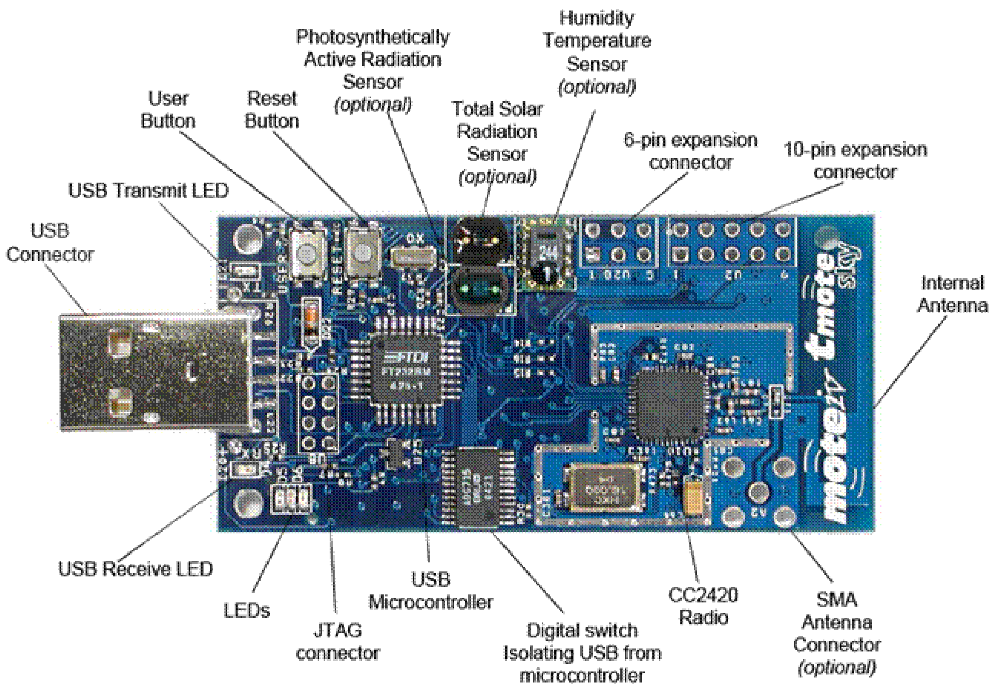

Photosynthetically Active Radiation Sensor (optional)
Humidity Temperature Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)

Figure 1.1: Tmote Sky sensor node [Moteiv Corporation 2006]

- peripheral extension connectors (analog and digital input/output ports, USB ports);
- an external storage device (a flash mempory chip or a microSD card);
- a power source (typically batteries).

The software is largely restricted by the resources of the microcontroller. Microcontrollers with very low energy consumption are typically used [Strazdins 2013], such as the 16-bit Texas Instruments MSP430 [Texas Instruments 2013] or 8-bit Atmel AVR [Atmel Corporation 2013] MCU families. Notable sensor network hardware platform examples that use MSP430 MCU include Tmote Sky, Zolertia Z1 [Zolertia 2013] and AdvanticSYS XM1000 [AdvanticSYS 2013]; notable examples that use AVR MCU include Arduino [Brock 2009]. In future, ARM Cortex-M0+ [ARM Holdings 2013] family can be expected to gain prominence, as it offers full 32-bit computation with active-mode energy consumption comparable to MSP430.

What can typical low-power MCU (like MSP430F1611 [MSP 2011] from MSP430 family or ATmega328p [atm 2011] from AVR family) offer to the sensor network programmer? Quite a lot. These devices feature a number of peripherals, such as general purpose input/output pins, multiple hardware timers, digitally configurable oscillator (determines the frequency of the MCU), watchdog, and others. Analog inputs can be sampled using the built-in ADC circuitry. There is hardware support for several popular digital data communication protocols: asynchronous 2-wire serial (USART), synchronous 3-wire serial (SPI) and 2-wire serial (I^2C). More advanced or newer MCU also include additional functionality, such as

Table 1.1: Comparative data of Texas Instruments MSP430 Series-2 microcontrollers and ARM Cortex M0+ based Freescale Kinetis microcontrollers. Representative sample from DigiKey catalog in March, 2013. Order quantity: 1000 units

MCU	**Code memory**	**RAM**	**LPM3 current**	**Price**
MSP430F2618TPM	116 kb	8 kb	1.1 μA	7.14 $
MSP430F2417TPM	92 kb	8 kb	1.1 μA	5.40 $
MSP430F2491TPM	64 kb	2 kb	1.0 μA	4.27 $
MSP430F2471TPM	32 kb	4 kb	1.0 μA	3.54 $
MSP430F233TPMR	8 kb	1 kb	1.0 μA	2.42 $
MSP430F2131TPWR	8 kb	256 bytes	0.9 μA	1.52 $[1]
MSP430F2001IPWR	1 kb	128 bytes	0.9 μA	0.51 $[1]

MCU	**Code memory**	**RAM**	**Stop mode current**	**Price**
Freescale Kinetis KL2	128 kb	16 kb	345 μA	4.19 $[2]
Freescale Kinetis KL1	64 kb	8 kb	345 μA	1.70 $[3]
Freescale Kinetis KL0	8 kb	1 kb	273 μA	0.78 $[4]

real-time clock (RTC) circuitry, or even universal serial bus (USB) support, but the previously described set of basic features usually remains included.

Table 1.1 describes the raw data for MSP430 and Cortex-M0+ MCU, while Fig. 1.2 visualises the correlation between memory capacity and price for MSP430 MCU.

The price of desktop-computer's CPU for the several recent decades has been mainly determined by its operating frequency and the number of cores. In contrast, the price of a very low power MCU strongly correlates with its memory capacity, including both code memory and RAM. For example, the seven MSP430 MCU selected for the sample has correlation coefficient 0.965 for the selected metric. (The correlation is not as strongly expressed for this sample if either only RAM size or only flash memory size is considered, therefore a combined memory usage index is used instead: flash memory usage plus RAM usage multiplied by 10. The selection of this specific index is justified by a rule of thumb described in [Levis 2012]: typical TinyOS applications use 10 times more code memory than RAM.) The correlation provides financial incentive to develop software that fits into the minimal amount of code memory and uses the minimal amount of RAM.

There is also another aspect of WSN hardware: the tremendous reduction of energy usage enabled by low-power modes. For some of the MCU the current consumption in sleep mode (shown in Table 1.1) is than 1000 times lower than the consumption in active mode (frequency-dependent & voltage-dependent; not shown in the table, but in order of several *milli*amperes, rather than *micro*amperes). This necessitates the development of software that spends as much of time as possible in the lowest-power mode possible. (Actually this is a simplification of the real

[1]Minimal order quantity: 2000 units
[2]Order quantity: 1 unit
[3]Order quantity: 96 units
[4]Order quantity: 490 units

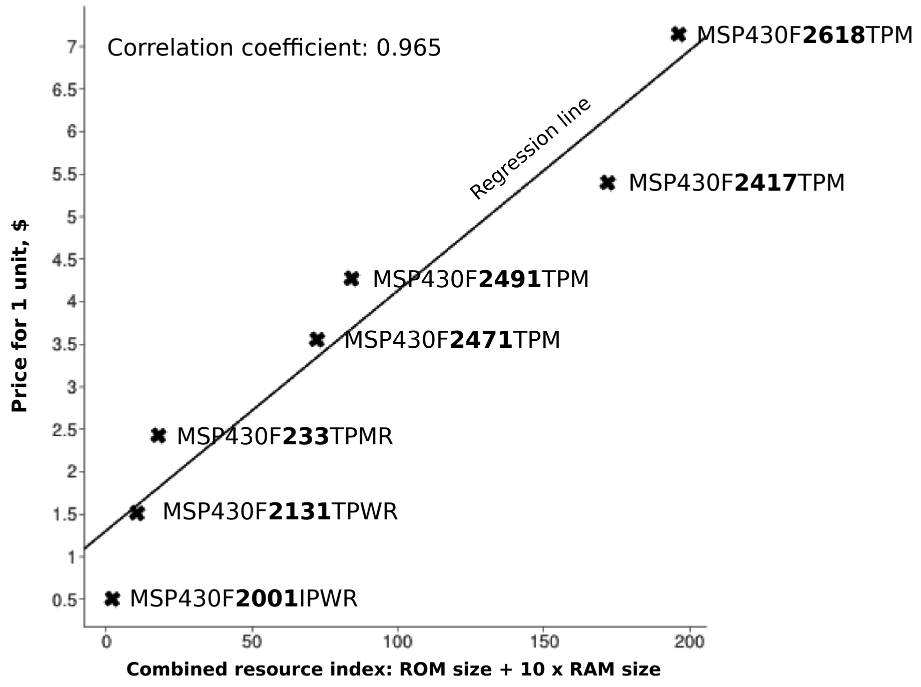


Figure 1.2: The price versus memory capacity of Texas Instruments MSP430 Series-2 microcontrollers. Representative sample from DigiKey catalog in March, 2013. Order quantity: 1000 units

situation; transitions between modes take non-negligible amount of energy as well, complicating the matter [Karl 2007].)

1.3 The programmers of sensor networks

This thesis is concerned with the *usability* of software for wireless sensor network application development (software such as operating systems or programming languages). ISO standards [Abran 2003] formally define usability as: “The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions” (ISO/IEC 9126-1, 2000). In this work, usability is generally understood as continuous rather than binary property. The usability of a particular software artifact is inversely proportional to the cognitive effort required to use it.

Since, by definition, usability is user-dependent property, in order to discuss usability of a particular software artifact, its users must be identified and described first.

1.3.1 The target audience of WSN programming tools

Besides professional software developers, in existing literature a target group named *domain experts* are identified as one of the users of WSN programming tools [Mottola 2011] [Bai 2009] [Miller 2009]. They include research scientists (for example, physicists and environmental scientists), civil engineers, and electronics hobbyists. The experts are believed to start with less programming background than traditional (desktop) application developers [Miller 2009]. Since the sensor network programming is unlikely to become one of their main activities, it is assumed that "the application domain expert will remain a perpetual novice, or, at best, an intermediate programmer" [Miller 2009]. However, "close collaboration between users, application domain experts, hardware designers and software developers is needed to implement efficient systems" [Romer 2004].

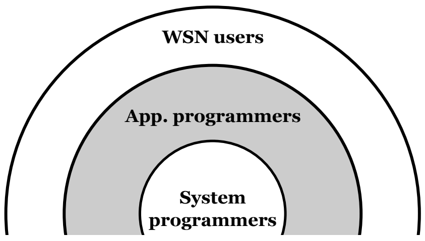


Figure 1.3: Euler diagram showing three levels of WSN users

In our opinion [Elsts 2012d] WSN application programmers is going to come from two fields: (1) advanced WSN users with programming experience; (2) professional application programmers with no WSN experience. Members of both of these groups lack system programming experience and have to learn a new programming languages in order to program sensor network applications. They form the grey area (Fig. 1.3) between operating system implementers and everyday users of WSN.

To use an analogy, the relation between these users and WSN programming professionals is similar to the relation between server operating system developers and server administrators. The administrators are not required to develop an operating system on their own; they are not even required to understand all the internal details; however, what they *are* required is to apply user-space tools provided by system programmers. The programmers, on the other hand, are required to develop tools that are easy to use for administrators and have comprehensive coverage of common administrative tasks. No coincidence that there is a number of domain-specific languages developed for system administration: for example, *bash*. In contrast, the WSN research community as yet has failed to develop a software tool with significant adoption in the target audience.

The task of a WSN programming tool or an approach is twofold. For the professional programmer it is to reduce her cognitive effort, the burden on her working memory, and to automate common tasks. For the novice programmer it is to make the programming feasible.

1.3.2 Assessment of computing courses in science study programs

The envisioned target audience of high-level sensor network programming tools comes from applied sciences. At the moment we are not concerned with computer science students (who can be expected to have proficiency in at least one general purpose programming language), but rather with persons with college-level training in other applied sciences. In order to evaluate the computing background of this target audience, we performed an examination of undergraduate study programs in University of Latvia (Table 1.2). We analyzed five study programs: Biology, Chemistry, Economics, Physics, Mathematics. (The study programs were selected in order to keep compatible with Kaplan's study [Kaplan 2004], described later in this section.)

We looked for courses in which either the basics of programming is taught or where programming skills are explicitly required. Courses that teach only applied informatics were not considered. We believe the results give some indication about the situation other areas of applied science as well, i.e. the domain in which WSN are useful as tools.

As the data shows, only students from mathematics and physics departments have computing as a part of their standard curriculum. The rest of them can take introduction to computing as a free choice course in another faculty – an option that can hardly be expected to be exploited by many undergraduate students.

Table 1.2: Computing course requirements for undergraduate science students (excluding computer science) in University of Latvia [LU 2012], school year 2011/2012

Program	Mandatory	Optional
Biology[1]	–	–
Chemistry	–	–
Economics[2,3]	–	–
Physics	–	+ [4]
Mathematics	+	+

Our findings are confirmed by an earlier study concerning teaching computing for undergraduate scientists in the USA [Kaplan 2004]. The curricula of nation's top ten liberal arts colleges were examined (Table 1.3). In universities of the USA, comparably low numbers of computing courses were required to be taken by science students (at least up to 2004, when the study was conducted).

We conclude that it is reasonable to expect that domain scientists won't have any significant training in computer programming beyond school level (with the

[1]A mandatory course in biometrics is offered, which mentions use of R software environment

[2]A related study program "E-business and logistics" is offered which features programming courses

[3]At masters level, one lecture in informatics course is devoted to Business Process Modeling Language (BPML)

[4]The programming course is listed as "mandatory choice" (B level), but is required to be taken to collect sufficient number of credit points for the semester

Table 1.3: Major requirements for computational courses in the United States of America [Kaplan 2004]

Department	Required	Allowed	No Mention
Biology	0	0	10
Chemistry	0	1	9
Economics	0	0	10
Physics	0	4	6
Mathematics	1	3	6

possible exception of physicists).

A separate question is whether school-level training can offset, on average, the lack of programming courses later on. This is an important question as programming is often taught in schools. However, we believe there are some indicators that school-level programming training is not a good predictor whether the transition between novice and non-novice programmers will be successfully made later (typically, at a university). For example, several respectable universities do not list school-level programming as a prerequisite for their introductory computer science class. ("Programming experience is good, but not necessarily required"[1] – Massachusetts Institute of Technology; "The course [CS106A] requires no previous background in programming"[2] – Stanford University; "There is no formal programming-related prerequisites for admission to 61A [Structure and Interpretation of Computer Programs]"[3] – University of California, Berkeley.) It is likely that these institutions believe that there is no critical knowledge gap between those whose have studied programming at school-level and those who have not. The author himself has only limited, one-semester exposure to basics of Pascal programming in school, giving a personal example of how formally limited school-level programming knowledge did not lead to delays when moving beyond novice programming level at the university.

1.3.3 Research about novice programmers

Novice programmers have been mostly studied in context of imperative general-purpose programming languages [Pane 1996]. Other areas, such as declarative domain-specific languages and novice programmers (the topic of this work) have "much room for investigation" [Pane 1996]. We speculate that it may be because differences between application domains are large, making programming approach well-suited for one domain useless in another. We examine the former studies, but acknowledge that they have only limited value in context of DSL for WSN.

[Robins 2003] mentions the generally accepted belief that it takes about 10 years to turn into expert programmer. Therefore, the gap between beginners and experts

[1] `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/Syllabus/`

[2] `http://www.stanford.edu/class/cs106a/handouts/03-course-placement.pdf`

[3] `http://www-inst.eecs.berkeley.edu/~cs61a/su13/about.html`

is huge. However, many of the issues that novices face are related explicitly to the nature of imperative programming. For example, “poor grasp of the basic sequential nature of program execution”, loops and conditionals (`for` loop in particular), the use of arrays, recursion, and especially “issues relating to flow of control”. Most of these issues are not present in a declarative domain-specific language that forbids recursion, such as SEAL (Chapter 3).

[Maloney 2008] describes a simple visual language Scratch and its “edutainment” (education + entertainment) use by 8 to 18 year old novice programmers. By analyzing hundreds of real programs created by this group, Maloney *et al.* have identified the usage of several concepts in these programs. The study identified that three concepts (loops, Boolean logic, and variables) out of seven concepts considered were increasingly prevalent between “second year” rather that “first year” students, i.e. they were valuable concepts, but with nontrivial learning curves.

In general, the novice programmer has to learn not only the features of the language, but also program design skills and problem-solving skills [Robins 2003]. Even if the language is trivial to learn (as are quite a few DSL), the skills are still required. However, if the user is a domain expert, she already has the domain knowledge, therefore the problem solving skills are reduced to solution formalization skills, which is arguably easier and has greater dependence on the language used.

[Neal 1989] stresses the fact that novice users often use examples to learn programming. The authors designed a programming IDE that made program examples easy to access and performed a test that required to develop an application using this IDE. Even though few of the test subjects actually copied example programs, the programs were used as visual aids. 15 of 16 subjects used the examples for ”more tricky parts”. Authors conclude that example-based programming is a promising approach, because “the response [of the audience] was extremely positive, and they found examples very helpful”.

1.3.4 Examples of novice programmers

In order to get a more concrete notion of our target audience, let us consider a few specific examples.

Philip the Physicist is working on models of radio propagation. He has a test setup consisting of number of spatially distributed outdoor antennas equipped with transceivers. The antennas stay in fixed locations during the length of the whole experiment, which can take several weeks. Philip would like to register environmental parameters (precipitation, air temperature) that affect signal propagation in his test.

The specialty of Elizabeth the Environmental scientist is microbiological plant diseases. In order to pinpoint the exact correlations between microclimate and prevalence of these diseases, she wants to record temperature, humidity and solar radiation in tree canopies during vegetation season. To get statistically significant data, she has to sample a number of trees; the number grows if several methods are applied to prevent these diseases.

Harry the Hacker is interested in so-called “smart houses” and is working on distributed energy consumption system in his free time. He wants to measure the energy consumption patters of each appliance, and automate some of them, so that less the energy has to be used in the peak hours.

Even though all these examples are fictional, they are based on real people, with whom the author is acquainted. And all of them could benefit from WSN in their work – even if they don’t know that yet!

The knowledge and skills of this small group are vastly different. Philip knows some FORTRAN for scientific computing, the basics of a few scripting languages, advanced mathematics, including some experience with MatLab, and maybe even some data-flow languages such as LabVIEW or Simulink. Elizabeth has studied some statistics, uses spreadsheets almost daily, and has been taught how to use at least one statistical package in the university (Section 1.3.2). Harry knows some C and assembly language, as well as basics of electronics, but tends to get confused if source code size is large.

What *is* common for all of them is that they are not only experts of their respective application domains, but also *novice programmers*. None of them has ever been a professional software developer; none of them has single-handedly written a program larger than a thousand lines of code; none is familiar with formal methods and tools of software engineering. Nevertheless, they are all technologically-literate, and willing to adopt and use new methods and tools. However, the learning curve of the existing programming tools can be an impassable barrier for a domain expert, because few of them have the willingness to spend significant time learning skills only marginally related to their main interests. Either they are able to learn how to use the tool “on the fly”, or the tool becomes next to worthless – no matter how well designed and intentioned it was.

1.3.5 Applicability of software engineering concepts to WSN

Software engineering aims to bring reliability and predictability in software development by attempting to make it more like other engineering disciplines. The methods of classical software engineering were created with large projects in mind: ”Software engineering concerns methods and techniques to develop large software systems. The engineering metaphor is used to emphasize a systematic approach to develop systems that satisfy organizational requirements and constraints.” [Van Vliet 2008]. One has to wonder whether software engineering has something to offer to WSN programmers at all, since WSN software is (1) relatively small, (2) authored by people from diverse backgrounds instead of software professionals, (3) usually developed as research prototypes with few organizational requirements and constraints. As examples show, even software that is apparently terrible in performing its intended tasks can be used as a base for WSN conference paper with large research impact [Langendoen 2006].

This question was tackled by [Picco 2010]. The author’s response is overwhelmingly positive: he concludes that “software engineering is unavoidable in WSN”.

Firstly, Picco reminds that WSN software, despite being small, is nevertheless quite complex. Secondly, he argues that focus on quality requirements is inevitable, if WSN ever can hope to become a useful tool for the society in large. It is of secondary importance whether WSN software will achieve the quality needed through formal methodologies, model checking, or other software engineering subdisciplines; the point remains that software engineering methods *are* needed. The above-mentioned [Langendoen 2006] article is a case in point: WSN projects that ignore basic software engineering rules are doomed to fail to meet their primary research goals.

The key idea in [Picco 2010] in the context of our work is his separation of two WSN programmer classes: the "end user" and the "WSN geek," who have very different concerns and skills. The objective of our work is to facilitate WSN programming for both of these groups; the main focus of SEAL is on the first group, and the main focus of MansOS is on the second group.

Although WSN applications come from diverse domains and show great variability in scale and requirements, they all have one domain in common: the domain of sensor networks. Two common approaches for domain-specific programming exist [Van Deursen 2000], the first being software libraries and component frameworks, the second: domain-specific languages (DSL). The first approach consists of building modules that support domain-specific concepts. Informally speaking, not only the so-called *middleware*, but also WSN OS falls in this category, including MansOS.

Domain-specific languages can be further divided by the means they are implemented: there are standalone languages and embedded domain-specific languages (DSeL). The second approach allows to facilitate domain-specific programming (compared to the library approach) by embedding a novel notation in an existing language. Several WSN programming approaches have chosen to go this way, for example Kairos [Gummadi 2005] and Pleiades [Kothari 2007]. A standalone language, compared with the other approaches, requires the largest *implementation effort* in contrast with the smallest *client effort*. By building a standalone DSL, the notation can be kept minimal and as relevant to the domain as possible. Compared with the library approach, DSL is also more amenable to automatic code optimizations and parallel execution, as it is easier to reason about.

1.3.6 Choosing appropriate level of abstraction

Modern software is characterized by great complexity; an opinion expressed in [Brooks 1987] is that "software entities are more complex for their size than perhaps any other human construct", because they contain very few redundant high-level elements. One of the main objectives of software engineering as a discipline is to fight against this complexity. This task can be approached by developing various *tools*. (In this context this term is understood broadly, and includes not only, for example, software development environments, but also *thinking tools*, such as software abstractions, and other means that facilitate software development, such as software libraries or languages.)

Let us consider two, somewhat contradictory, ways how the complexity in software development can be reduced: the first is to raise the level of *abstraction* provided by the tools in question, and the second is to raise level of *concreteness*. Consider these three very simple software libraries as an example of tools with different levels of abstraction:

- The first library for arithmetic operations implements a function that calculates the sum of an arbitrary rational number and 2.
- The second library implements a function that calculates the sum of two arbitrary rational numbers.
- The third library implements a function that calculates the sum of two arbitrary p-adic numbers.

The first library is too restricted to be useful for large number of applications. The last one may be too generic for many applications, and may require additional learning overhead, as non-mathematician users are frequently not familiar with p-adic numbers (they are a generalization of rational numbers). The second one in this example clearly has the widest scope of application.

One can conclude that it is not enough to offer the most abstract tools (in particular, thinking tools), as the tools themselves will become hard to comprehend, especially for practically-minded users. It is not enough to offer the most concrete tools either, as in this case all possible applications of these tools will be tied to one particular problem; the tools will lack generality and flexibility. Abstraction is beneficial to software reuse [Krueger 1992].

Therefore we argue that it is important to get the *right* level of abstraction. The appropriate level is determined by the users and their previous experience. In our opinion, the scientists from experimental and applied sciences, who deal with hands-on experiments using physical world objects, are less likely to have preference towards the abstract, detached thinking typical of theoretical computer scientists and software developers, whose work focuses on the purely virtual. This implies that the software abstractions presented to domain experts should be with concrete real-world counterparts when possible. (A similar suggestion was distilled from novice programming literature by Pane and Myers in their survey article [Pane 1996].)

1.4 Operating systems

1.4.1 Design overview of sensor network operating systems

The tasks of a traditional operating system include managing resources, protecting access to them and allocating these resources to multiple users, as well as support for concurrent execution of several processes and communication between them [Tanenbaum 2006]. In contrast, a sensor network operating system typically has to serve only a single application belonging to a single user. The severe hardware limitations means that it may be impossible (or at least undesirable) to implement all the functionality of a typical traditional operating system. However, there are some specific tasks that have paramount importance in wireless sensor networks, namely, concurrent processing and energy management. Sensor network operating systems should have support for these tasks.

Some common services that a WSN OS should offer to the user include: networking (routing, forwarding, MAC layer, time synchronization); sensor/actuator/-transceiver access protocols; nonvolatile memory access; a library of common functions (such as calculation of cyclic redundancy check error detection codes); time accounting; setting system policies (such as policies for energy usage), and so on [Karl 2007].

The operating system should also support *reprogramming* [Wang 2006]. Since a sensor network typically consists of a large number of devices deployed in a remote environment, it should be possible to reprogram them remotely. Implementing reprogramming is nontrivial, because these devices are battery powered; the application code should be as small as possible in order to save energy required for data transfer in the reprogramming process. One idea is to run the application on top of a virtual machine; the size of the bytecode for a virtual machine tailored for WSN typically can kept much smaller than the size of equivalent machine code for a MCU [Levis 2002]. However, being able to use only pre-defined bytecodes limits flexibility of applications. Therefore less exotic alternatives should be supported in WSN OS as well, such as partial reprogramming: a process where only part of the binary code is replaced, rather than all of it.

In contrast to desktop systems that are executing on platforms with hardware support for multiple access levels, the distinction between "user" and "system" execution contexts is usually not as strongly pronounced in sensor network operating systems. In fact, both "user" code and "system" code may access the underlying hardware directly. Furthermore, the system and the user code are usually not put in distinct addressing spaces. The sensor network "operating system" code does not exist separately from the "user application" code; they are compiled separately, but linked together to produce a single, optimized binary image. (Further in the text, we sometimes use the term "application" in order to refer to the final binary image of an executable, even though it includes not only user code, but also operating system code. This is justified by the fact the the image is sometimes built and optimized in application-specific way, as discussed below.)

In fact, it may be disputed whether the term "operating system" applies to sensor network software like TinyOS [Levis 2004b] at all; Karl and Willig [Karl 2007] mention that "perhaps the term execution environment is the more appropriate one". However, in this dissertation we retain the traditional terminology of the WSN research community, and call TinyOS (for example) an operating system.

The fact that the whole operating system may easily be recompiled whenever a user application is recompiled can be used to adjust the operating system for the needs of each particular application. Only the components actually used by the application have to be included in the binary image; furthermore, extensive compile-time and link-time optimizations on these components become possible. However, from all of the operating systems discussed below, only TinyOS adopts this design idea to the fullest; the implementation of this idea in this OS required development of a special programming language nesC and a compiler for this language.

1.4.2 Execution models of sensor network operating systems

The execution model of a software artifact influences not only energy-efficiency, but also determines the programming model that is appropriate for this software artifact. Therefore “execution model” and “programming model” are sometimes used interchargeably in this dissertation.

A purely sequential model of execution is inappropriate for sensor network software. Data may come in from “arbitrary sources at arbitrary points of time” [Karl 2007]; some data may have higher priority. Several approaches how to implement concurrency in WSN OS exists:

- **Event-based.** The operating system reacts on hardware events, such as reception of a radio packet, data from a sensor, or expiration of a hardware timer. The event handler may schedule the so-called “task” (callback function) and return. The task does the actual information processing; e.g. looks what it inside the radio packets and decides which application-specific actions to perform. Tasks are *run to completion* and cannot be interrupted by other tasks; they can be interrupted only by events.
- **Preemptive multithreading.** In this model, different services and applications correspond to different threads of execution. The hardware signals when the currently active thread should be replaced with another one. It may happen either when the time-slice of the current thread has expired, or when an event has happened that should be processed in some other thread. In any case, the so-called “context switch” takes place: the state of the current thread is stored to memory, and the state of the thread that is to be run is retrieved from memory. Each thread must have its own stack space preallocated during program initialization; thread-local context is stored in this stack space.
- **Cooperative multithreading.** This model is similar to the previous one, except that threads cannot be preempted by other threads, only by event handlers. However, a thread can give up its execution context voluntarily.

The initial idea of WSN research community was to go for event-based OS design at all costs, because it provides the maximal efficiency. For example, [Karl 2007] (first published in 2005) advocates event-based programming model as “the only feasible way”. However, the book based this view on a single evaluation [Li 2003] that compared TinyOS with an existing, relatively “heavyweight” embedded operating system eCos. Later development of WSN-specific operating systems with multithreading (such as Mantis [Bhatti 2005]) forced a reevaluation. Even though the current multithreading OS are still not as energy-efficient as TinyOS [Duffy 2008] [Lajara 2010], the prevailing mood in the community has changed; energy efficiency gains that compromise programming efficiency now are shunned upon. In particular, [Duffy 2008] shows that threaded operating systems are preferable in many WSN application areas.

1.4.3 The current tradeoff between efficiency and learnability

One way to characterize the current WSN system level software design space is by plotting it across two dimensions: the first being efficiency, the second – how easy the system is to learn (Fig. 1.4). For example, TinyOS puts efficiency and reliability over simplicity [Levis 2012] and is marked by a steep learning curve [Levis 2012] [Cao 2008] [Bhatti 2005]. It can be put approximately at one corner of the design space. In contrast, Arduino [Brock 2009] allows to write a few typical applications with remarkable simplicity [Faludi 2010] [Brock 2009]. The time required to start using their software libraries is short, as a relatively simple IDE and programming examples are provided; however, the feature set is severely limited (no multithreading, no OS-level support for low power modes and so on), and the energy efficiency of the system is not as great [Wheat 2011]. It can be put at the opposite corner.

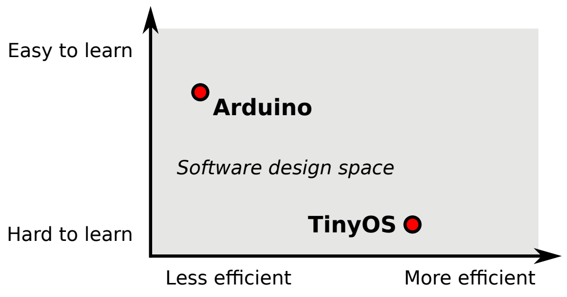


Figure 1.4: Sensor network software design space approximately projected to the dimensions of efficiency and learnability (qualitative estimation)

A yet unreached goal in WSN OS design is to create a WSN OS with efficiency comparable to TinyOS and usability comparable to Arduino. This dissertation is an attempt to move in that direction. Admittedly, classical WSN OS are inherently more complex, because they support networking and take distributed processing and energy efficiency as core issues, while Arduino software libraries do not. For example, Arduino only supports sequential execution model, which is insufficient for WSN, as was argued above. Therefore, equally high usability for WSN OS may be impossible to fully reach without moving to higher abstraction layers. However, higher-level interfaces come at the price of decreased flexibility, which not all are willing to pay.

1.4.4 Sensor network operating system examples

In this survey of the existing WSN OS we have included four well-known research-oriented WSN OS together with Arduino. A brief timeline of the academically-oriented WSN OS analyzed in this thesis is shown in Fig. 1.5.

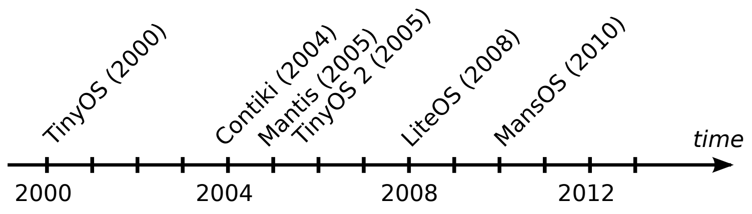


Figure 1.5: Timeline of wireless sensor network operating systems (by the date of the first publication)

1.4.4.1 TinyOS

TinyOS is a seminal sensor network OS that has very high impact in the research community. A primary goal of TinyOS was to enable and accelerate WSN research [Levis 2004b].

The initial hardware supported by TinyOS was a very limited 8-bit MCU with 8 KB flash, 512 byte RAM, and bit-level radio [Hill 2000]. Although later the focus of the system changed to more typical WSN hardware, efficiency has remained a primary concern.

TinyOS is written (and requires applications to be written) in a specific programming language: *nesC*. The need to learn a new language makes using the OS more complicated for non-expert users. Additionally, there are more factors that contribute to the steepness of TinyOS learning curve: novel software abstractions present (static virtualization, parametrized components, etc.), the fact that concurrency is fully exposed to the user, and high generality and modularity of the OS itself.

Application building was often very hard with TinyOS 1 [Levis 2005]. TinyOS 2 was an attempt to ameliorate these difficulties in two ways: first, by allowing build applications on top of service distributions, second, by increasing the reliability of the system. Even a trivial sense-and-send application was shown to behave incorrectly in TinyOS 1 [Levis 2005].

The initial design of TinyOS included support only for run-to-completion tasks. Multiple ideas how to integrate threads in TinyOS were soon proposed, until TinyOS 2.1 included support for TOSThreads [Klues 2009]. TOSThreads do not break TinyOS concurrency model, because the kernel is still event-based. Kernel tasks are not interruptible by user threads, basically meaning that two priority-level scheduler is in place.

In contrast to many early attempts to move TinyOS out of the lab and apply to real-world deployments, in more recent years academic deployments of TinyOS powered sensor networks often turn out to be a complete success [Levis 2012]. Nevertheless, finding new contributors to the core code itself turned out to be challenging. The authors of TinyOS admit that both over-generalization and too finely grained components add to the difficulty of understanding and modifying the OS itself. For example, the code for CC2420 radio driver is distributed across 40 source files, making it very hard to build a mental model of it.

1.4.4.2 Contiki

Contiki is a lightweight operating system created at the Swedish Institute of Computer Science [Dunkels 2004]. Contiki applications are written in plain C, support dynamic loading and unloading of components, and run on top of many popular WSN hardware platforms (for example, TelosB and Zolertia Z1 [Zolertia 2013]).

Contiki is recognized for its unique execution model, which is a form of cooperative multithreading: on top of event-based kernel lightweight cooperative threading primitives are executed, called *protothreads* [Dunkels 2006]. Although not without limitations (see Section 2.4), protothreads is a simple and elegant alternative both to event-driven application code and to preemptive multithreading. Protothreads are widely used in design of Contiki; most of its stateful system services (networking etc.) are implemented as protothreads. The system also provides an interface for writing platform-specific preemptive multithreading schedulers, although this approach appears to be much less popular. Such a scheduler is not present in the distribution of Contiki itself.

Contiki is also renowned for its timely inclusion of resource-efficient implementations for IPv4 and IPv6 networking, bringing the vision of Internet of Things closer. There is also an option to use a more lightweight communication stack called RIME. Contiki includes support for low power listening and energy efficient (low duty cycle) data forwarding, as well as many other typical networking primitives.

Contiki has quite efficient code architecture that separates the MCU layer from hardware platform layer. MCU family, such as MSP430 and AVR, code is put separately from platform-specific code (like TelosB-specific code, Zolertia-specific etc.). However, some code duplication is present (see Section 2.2.3).

Contiki adopts "most of functionality is included by default" strategy, which leads to large binary code size (Section 2.6.2). However, for some components the user may replace the default implementation with a version that simpler or more appropriate to his application (for example, the default Contiki-MAC protocol can be replaced with a dummy "null" MAC protocol). Even more, networking subsystem can disabled in total; IPv4 and IPv6 support in the network stack can also be disabled separately.

1.4.4.3 Mantis

Mantis OS is a WSN OS created at University of Colorado at Boulder. The date of the first Mantis publication is 2005 [Bhatti 2005].

Multithreading is a key design feature of this WSN OS. The design goal of multithreading in Mantis was to naturally interleave processing-intensive tasks (such as data encryption or compression) with time-sensitive tasks (such as network communication). The authors stress that manual partitioning of complex tasks in smaller time slices is not trivial and sometimes require knowledge about the semantics of the algorithm. If the programmer is merely porting the processing-intensive algorithm to a WSN hardware platform, she might lack the necessary knowledge. At the same time, failure to partition the tasks in small-enough slices might lead to contention of memory buffers and dropped data packets, because the consumer task unable to execute active while the producer task is running to completion, as in TinyOS (see also Section 2.4.1). The programmer also has to ensure that no components use busy-waiting polling and infinite loops. In contrast, if preemptive multithreading is used, the scheduler is going to automatically time-slice between threads. Thus, usability is the primary motivation of this Mantis feature.

The scheduler can handle threads with multiple priority levels and provides round-robin based scheduling within a single priority level. The scheduler also handles sleeping when all threads are idle and have explicitly enabled power save mode. The questionable choice to make this the non-default behavior was motivated by the desire to keep compatibility with UNIX `sleep()` behavior.

Mantis is designed to work on multiple hardware platforms. In particular, application code can also be compiled to run natively on x86 architecture. This allows multimodal sensor network prototyping, where part of network consists of hardware sensor devices and part consists of virtual nodes simulated on the PC.

1.4.4.4 LiteOS

LiteOS is a relative latecomer to the WSN OS stage (first publication in 2008 [Cao 2008]) and takes the ideas already seen in Contiki and Mantis to the logical extreme, making the OS as much UNIX-like as possible. Being easy-to-use is declared as one of LiteOS primary design goals, approached through features like (1) a distributed file system as an abstraction for the sensor network, (2) dynamic loading of separate "applications" (i.e. threads), (3) advanced event logging, dynamic memory support and other features.

The trump card of usability for LiteOS is arguably the reuse of concepts from UNIX operating system, which is familiar to many of the potential WSN programmers. LiteOS authors point out that this reduces the steepness of the learning curve, as existing knowledge of the system programmer is reused. Namely, the shell, filesystem, and threads are familiar concepts, as is the C programming language itself. However, the suitability of these concepts to WSN is not always clearly shown. In particular, there is no evidence that the "network as a file system" abstraction is easy to grasp, especially because it often diverges from the tacitly assumed behavior. For example, a *mounted* directory usually guarantees data availability – something that might be impossible to achieve in WSN with their characteristically unreliable communication. Furthermore, a seemingly innocent operation like simple file access might in fact require relatively large energy usage, causing the lifetime of the network to be significantly reduced. Other abstractions are not without faults as well. LiteOS shell command `ps` lists different threads as different applications, which is a questionable choice, because UNIX processes are typically marked by lack of shared memory, while WSN OS threads often share a global state. The LiteFS file system uses POSIX-like (but not POSIX-compatible!) API arbitrary extended with WSN-specific functions, for example, `fsearch`, `fcheckEEPROM` etc.

LiteOS runs on a single hardware architecture: Atmel. The portability of the system is low, because inline assembly is frequently used.

To demonstrate the usability of LiteOS, the authors have ported a number of TinyOS applications to it [Cao 2008]. An interesting observation is that the number of threads required for each application is small – from the 20 applications ported, only eight use more than one thread, and only two use the maximum of three threads (see also Section 2.4 on how to exploit this fact). (We acknowledge that most of these applications are trivial.)

1.4.4.5 Arduino

While not generally considered a WSN platform, Arduino can be used to successfully build sensor networks [Faludi 2010], and Arduino software libraries often fulfills the role of an operating system. Therefore it is included in this survey twice: first here, among operating systems, and then in the next section, among programming approaches.

What exactly is Arduino? Just a "low-cost platform for embedded computing" [Brock 2009]. Arduino, which is printed circuit board with an ATmega microcontroller, surrounded by several digital and analog I/O pins, can be easily turned into a sensor device by attaching external inputs to the pins. Arduino can be interfaced to, for example, "switches, sensors, lights, and motors" [Brock 2009].

The popularity of Arduino software can be explained by two reasons: the popularity of its hardware; and the large, friendly and enthusiastic user community. There are more than 300 000 hardware units sold [Arduino Team 2012]. Both Arduino software and hardware are perceived as novice user-friendly.

While the site claims that "the Arduino board is programmed using the Arduino programming language, which resembles C with a touch of C++," the Arduino programming language is in fact a subset of C++. Arduino is built around the Wiring project, which is an electronic version of Processing programming language. Arduino IDE is derived from the Processing IDE. (See also Fig. 3.7.)

Efficiency is not a primary concern, neither in Arduino software nor in hardware. First, the MCU is Atmega, which is not as energy efficient as the ultra-low power MSP430. A typical Arduino also requires 5 V input voltage, making it harder to power the board in field conditions.

Regarding software, the user-friendly, but inefficient design is evidenced by an example from [Wheat 2011]: a simple call to `digitalWrite()` function takes 66 MCU cycles in average. For comparison, in MansOS the same code (which is a C preprocessor macro instead of a function) takes 8 cycles: 4 for setting a pin in output mode, and 4 for setting its value.

Support for low-power mode management is not included in the Arduino software libraries. If a user wants to put the system in a sleep mode, she not only has to do it manually, but also to use API provided by the manufacturer of the MCU.

Arduino applications are single-threaded and based on a very simple and rather restricted execution model. The user has to provide code for two functions: initialization function and a "loop" function. The latter is called repeatedly in an infinite loop in the Arduino core library. While simple to understand, the model is not always as simple to apply: for example, all external events must be handled by polling in the loop function. If the application has to implement, for example, handling of two concurrent data streams with various QoS requirements (reliability, delay tolerance), writing the user code become far from trivial. At the same time, requirements like these are nothing untypical for WSN applications.

All of this allows to conclude that while remaining highly usable in general, Arduino puts simplicity over usability in this, specific, WSN context.

1.5 Programming approaches

Survey of several of existing WSN programming languages, libraries, and middleware is given in Table 1.4 (ordered alphabetically). Following the practice of several existing survey articles [Rubio 2007] [Mottola 2011], all these software artifacts together are called *programming approaches*.

Compared to existing work, in what ways this survey is different? One drawback of Rubio *et al.* [Rubio 2007] is that it is very concise and do not list details of each approach. More importantly, several more practically-oriented qualities such as the ones described in the last six columns of Table 1.4 were not analyzed in either of the two surveys. In general, the qualities we were looking for were specifically determined by the goals and scope of this dissertation.

Several of these approaches are classified as so-called *macroprogramming* [Mottola 2011]. This term refers to WSN programming "at large", at the level of whole network or its region, rather than at level of a single node. It is contrasted with "microprogramming", i.e. single node programming.

Although majority of the approaches listed here recognize the importance of powering application experts with the tools to program WSN, only a few of them pay attention to user evaluation studies, IDE, and visual programming options. Their focus, understandably, is different; however, if we expect a programming approach that ir popular in the real world to appear, user input and real-world evaluation results is of critical importance. SEAL (Chapter 3) is also included in the table.

Table 1.4: WSN programming languages and models

Approach	**Type**	**Scope**	**Universality**	**Made for novices**	**Simple concepts only**	**Visual programming**	**IDE**	**Usability evaluation**	**Real-world applications**
Abstract Regions	API	Regional	+	+/−	+	−	−	−	−
Abstract Task Graph	DSL	Global	+	+	+	+/−	−	−	−
Arduino	API	Node-local	−	+	+	+[1]	+	?	+
Erlang	API	Node-local	+	−	−	−	−	−	−
Flask	Embedded DSL	Node-local	+	−	−	−	−	−	−
Kairos	Embedded DSL	Global	+	−	+/−	−	−	−	−
Logical Neighborhoods	DSL	Regional	−	−	+	−	−	−	−
makeSense	Embedded DSL	Global	+	+	+	?	?	−	+
nesC	Embedded DSL	Node-local	+	−	−	−	−	−	+
Pleiades	Embedded DSL	Global	+	−	+	−	−	−	−
PySense	Embedded DSL	Global	+	−	+	−	−	−	−
Regiment	DSL	Global	+/−	+/−	−	−	−	−	−
SEAL[2]	**DSL**	**Node-local**	+	+	+	+[3]	+	+	+

[1]Using BlocklyDuino [Lin 2012]
[2]Described in Chapter 3
[3]Using SEAL-Blockly (Section 3.3.5)

Table 1.5: WSN programming languages and models (continued)

Approach	**Type**	**Scope**	**Universality**	**Made for novices**	**Simple concepts only**	**Visual programming**	**IDE**	**Usability evaluation**	**Real-world applications**
SenQ	DSL	Global	−	−	+	−	−	−	+
SensorBASIC	Embedded DSL	Node-local	+	+	+	−	−	+	−
SNACK	DSL	Node-local	+	+/−	−	−	−	−	−
Snlog	DSL	Global	+/−	−	−	−	−	−	−
Squawk	API	Node-local	+/−	−	−	−	+	−	+
TeenyLIME	API	Global	−	−	−	−	−	−	+
TinyDB	DSL	Global	−	−	+	−	−	−	+
TinyScript	DSL	Node-local	+	+	+	−	+	−	−
TinySOA	API	Global	−	+/−	+	−	+	+	+
WASP	DSL	Node-local	−	+	+	−	+	+	−

Column titles of Table 1.4:

- **Type** – domain-specific language, embedded domain-specific language, or library API?
- **Scope** – node-local, regional, or macroprogramming focus? (Note: the margins often are somewhat blurred)
- **Universality** – universal or restricted applications?
- **Made for novices** – does the model: (a) declare domain experts as their target audience, and (b) for a typical use case[1] does not require to implement advanced distributed algorithm (as in Abstract Regions) or understand internal workings of OS-level concepts (as in SNACK).
- **Simple concepts only** – no advanced programming concepts used? (For example, functional reactive programming, logical programming, functions as first class values are counted as advanced concepts)
- **Visual programming** – applications can be developed by manipulating program elements graphically?
- **IDE** – integrated development environment present?
- **Usability evaluation** – user evaluation studies performed?

[1]See e.g. [Strazdins 2013] for survey of typical applications

- **Real-world application** – real-world use cases known to us? (Either from the corresponding publications or by some other means.)

In the table, '+' means "fully supported", '−' means "not supported", '+/−': "partially supported", and '?' is "unknown".

The goal of **Abstract Regions** is "to simplify application level design by providing a new set of programming primitives" [Welsh 2004] for addressing, data sharing, and aggregation among nodes in the network. Multiple implementations of a region is possible and several kind of examples are provided by the authors (e.g. where region is the single-hop neighborhood of a node). Their vision is to simplify programming for domain experts; however, the identified "core difficulty" they want to overcome is the difficulty of expressing global interaction in terms of complex local actions. This view is not well supported by the current real-world WSN applications, where complicated processing is an exception rather than a norm.

Abstract Task Graph (ATaG) [Bakshi 2005] is a data-driven method for specifying sensor network programs using mixed imperative-declarative syntax. It uses abstract tasks, abstract channels, and abstract data as main primitives. The focus is on macroprogramming and information flow in the network. The model is architecture independent and network independent, although "network-aware". ATaG features two-phase code development: first the declarative part is specified graphically, then programmer inputs code in the generated template. In our opinion, the additional complexity of separating the two-phases (the programmer has to learn and use two completely different programming paradigms) is not outweighed by the gains in cohesion and conceptual simplicity. The ATaG programmer also has to know a traditional programming language, such as C or Java, to be able to program the imperative part.

The primary task of **Arduino** software libraries is to support embedded programming on Arduino boards, but they can be used to program WSN as well [Faludi 2010]. In contrast to other approaches, its roots are not in the WSN research community, but rather from Processing environment, which has the objective to teach basics of computer programming to nonspecialists [Reas 2003]. Therefore, its usability is good (empirically supported by the large number of Arduino users), while compatibility to WSN-specific requirements is limited. Also, users cannot be completely shielded away from low-level memory access details, because Arduino uses C++ as the programming language.

There also is an attempt to adopt **Erlang** programming language for sensor network programming [Sivieri 2012]. Erlang is a functional language with many practical applications in networking domain; the strength of Erlang is its good language-level support for concurrency. The attempt is still in early stages and the technical aspects are described only in cursory detail in the published work. The main problem of this approach is high storage and memory footprint, even though a custom, stripped down version of Erlang interpreter was developed for sensor networks. The exact numbers are not given, but the author himself describes it as "quite high" (order-of-magnitude estimate: megabytes).

Flask [Mainland 2008] is an attempt to bring functional reactive programming to sensor networks. The authors use a real application (Ecuador volcano monitoring [Werner-Allen 2006]) as their showcase; however, the intuitiveness of the syntax and concepts used to domain experts is dubious at best. Being a domain-specific language embedded in Haskell, Flask takes a lot of its expressive power from higher level abstractions that are built using combinators. Due to this high level of abstraction and amount of prerequisite knowledge required from the user, Flask is not applicable in the context of novice programmers.

Kairos [Gummadi 2005] focuses on expressing global behavior of distributed computation, rather than on node local and regional abstractions. Kairos is implemented as an extension to Python (traditionally considered novice-friendly language). Kairos provides three simple abstractions: (1) node abstraction – manipulate nodes and lists of nodes; (2) neighborhood abstraction – one hop neighbors of node; (3) remote data access abstraction – the ability to read variables from named nodes. The tasks Kairos article lists are either not suitable for novice programmers (building a routing tree and localizing sensor nodes) or not frequently met in real-world WSN deployments (distributed vehicle tracking – not met in the sample analyzed in [Strazdins 2013]).

Logical Neigborhoods (LN) [Mottola 2006] is a programming abstraction that allows to group neighboring nodes not only by physical, but also by logical attributes. The abstraction is implemented in SPIDEY language proposed by the authors. The support of Wireless Sensor and Actuator Networks with multiple sinks is stated as their main goal; therefore, in contrast to the above described WSN programming models, LN is limited rather than universal in scope. LN uses two-phase approach to programming – templates and instantiation. Informally, they correspond to *what* is useful to the application, and *how* it can be used. In contrast to ATaG, both phases are implemented in the same language. In our opinion, although the concepts used by LN (virtual neighborhoods, virtual sensors) have no real-world counterparts, they should be simple enough for domain experts to grasp.

makeSense [Casati 2012] is a new development that explicitly declares increased industrial adoption of WSN a problem they want to address. The authors name lack of unification in WSN software and lack of its integration with business processes as the two main problems. The result of the project is expected to be “not another macroprogamming language”, but a framework for integrating existing solutions. Event though our goals are closely overlapping, the target audience of makeSense project is different: users familiar with Business Process Modelling Language (BPML).

nesC [Gay 2003] is the language of the most popular WSN operating system: TinyOS. As such, it has many successful real-world applications. However, the unsuitability of nesC to novice users has been long lamented by WSN research community. Using this language not only requires understanding TinyOS internals, but also using split-phase programming model (which is arguably harder to grasp), because the language and OS targets event-based execution flow. As our evaluation show (Section 3.5.1) it leads to high cyclomatic code complexity.

Pleiades [Kothari 2007] is an extension of C language for WSN macroprogramming. The Pleiades compiler generates code for individual motes, and takes care of distributed locking and "serialization" of code execution. However, in contrast to some other global WSN programming approaches (e.g. data query languages), access to node's single-hop neighborhood is still possible. Among the features of Pleiades compiler is the support for program migration from node to node. In our opinion, Pleiades is one of the most promising WSN macroprogramming approaches up to date, but not very well suited for novice programmers because C is used. Among the drawbacks of this embedded DSL is the lack of node failure handling, making the version of the language reported in the paper unsuitable for practical usage.

Among the mainstream programming languages that had been applied to sensor network programming not only Java, but also Python is prominent. **Pysense** [pys 2013] is a sensor network macroprogramming approach that extends Python language with custom *decorators*. One of main design goals of Pysense is to reduce energy consumption in data transmission by supporting intra-network data processing, as well as by allowing to cluster nodes in regions. The Pysense is "still incomplete to be used in real-world applications".

Similarly to most of the approaches described here, **Regiment** [Newton 2007] aims to solve the complexity problem of WSN programming. It explicitly mentions accessibility to nonexpert as the vision of WSN programming tools. However, this macroprogramming framework is based on functional reactive programming, evaluated for distributed event detection scenarios (not among the most popular real-world WSN applications), and uses highly abstract concepts (e.g. sensor network state is represented as time-varying signals). All of this makes Regiment better suited for the WSN programming professional than the domain expert. Regiment targets a specific application class: "spatiotemporal macroprograms"; however, the paper states the belief that it is possible to "make macroprogramming viable for majority of applications". The authors of Regiment also state an important point that it is apparently impossible to completely abstract away performance implications of WSN programs.

SenQ [Wood 2008] is a distributed query system for WSN, similar to TinyDB. The infrastructure of SenQ provides support for multiple distributed data streams in the network, while the application logic is specified using a declarative query language SenQL. This approach is too limiting to be considered a direct competitor to our target software. However, in contrast to most of approaches described here SenQ was used in real-word WSN deployments [Selavo 2007] and prototypes [Wood 2006].

SensorBASIC [Miller 2009] explicitly addresses the problem of bringing WSN to domain experts by applying a dialect of BASIC language to WSN programming. Somewhat counterintuitively (c.f. Section 1.3.3), the authors state that imperative loop oriented programming was the most successful for the users tested. The efficiency of BASIC interpreter on WSN motes leaves a lot to be desired; however, the authors counter this overhead by claiming that, first, WSN applications are not computationally intensive; second, that the code can be compiled to run natively. In any case, the size of the interpreter is relatively large – 38 kb in flash: impractical for some popular motes as Tmote Sky. This is one of the few articles where user evaluation results were presented. The results were favorable. However, regrading domain experts as users: only a small-scale anecdotal study was performed, using only two participants. The authors explicitly encourage more work in this direction. In general, our goals are similar to the goals of [Miller 2009], but the details are very different – we use declarative instead of imperative programming, and a novel, rather than existing programming language. The authors argue that BASIC provides easy-to-understand execution model; we respond that programming in declarative languages lessens the need to understand the execution model at all. Programming in a correctly designed restricted, declarative language makes it harder to write incorrect or inefficient code compared to imperative language, where the programmer has to explicitly describe program flow.

SNACK is an early, but impressive work in this field that presents "a new configuration language, component and service library, and compiler that make it easier to develop efficient sensor network applications" [Greenstein 2004]. It does not introduce new programming paradigms *per se*, but is a tool to build systems from blocks implemented either in SNACK itself or in lower level languages. The idea is to raise abstraction level by avoiding working with low-level individual components. The target audience of SNACK consists of three types of programmers. First: system designers, who should be nesC and operating system experts; they create components to be used by the others. Second: service designers, who bind the components together in service libraries; they should be experienced WSN users. Third: application programmers, whose tasks "should be simple enough for scientists to do, powerful enough to support a wide range of applications, and generate efficient enough programs". However, even for the application programmer significant entry barriers are present. The concepts used in SNACK (e.g. transitive arrows) are sometimes not straightforward. The syntax is based on the syntax of the Click modular router and is in some aspects similar to Perl syntax. In our view, it could be cleaner: it features large proportion of non-alphanumeric characters, and looks rather cryptic at times. Our other points of critique for SNACK are: lack of usability study; lack of validation of the solution in real deployments; TinyOS-specificity of the solution, as it depends heavily on the way components in TinyOS are developed and used. Finally, the components used in SNACK are generally too low level to be directly usable by domain experts.

Snlog [Chu 2006] [Chu 2007] [Tavakoli 2007] shows how to build several WSN services and applications fully declaratively, thus introducing a new WSN programming paradigm. The language is a dialect of Datalog, and has the look-and-feel of Prolog. The application code written in Snlog is much shorter than code developed using less high-level approaches. However, we doubt that it is much simpler to write and read for the nonprogrammer, because studies show [Pane 1996] that Prolog is "notoriously difficult to grasp for novice programmers".

Squawk [Simon 2006] is a Java virtual machine that allows to program Sun SPOT (Sun Small Programmable Object Technology) motes that run "Java directly, without operating system" [Smith 2007]. Even though the Sun SPOT hardware is characterized as "small", it still has much more resources (512 kb RAM, 4 Mb flash memory) than a typical sensor network mote like Tmote Sky. These resources are required to run Squawk (80 kb RAM usage, 270 kb storage usage). Therefore Squawk is not directly applicable to the context of this work, in which more resource constrained devices-are considered.

TeenyLIME [Costa 2007] is a WSN middleware founded on the notion of tuple space and is focused on sense-and-react applications. The notion of tuple space is understand as distributed repository of tuples; a tuple is a list of typed fields. TeenyLIME applications manipulate the tuple space (e.g. remote sensor readings) using TeenyLIME API. The API is implemented as a library in nesC, therefore its suitability for direct usage by novice programmers is rather doubtful. Furthermore, all TeenyLIME operations are asynchronous and split-phase, arguably complicating writing the application code. Finally, to manipulate the tuple space, pattern matching is used – the novice programmers might not be perceive it as conceptually simple.

TinyDB [Madden 2005] is a query processing system for WSN focused on efficiently sampling and aggregating the data. It is a completely global programming approach with no option to address sensor nodes individually. Support for novice programmers was not stated as explicit goal, but is a possibility taking into account the simplicity of application logic implemented as TinyDB queries. All in all, it is a rather limiting approach to be considered a direct competitor to our target software. The code of TinyDB is comparatively large (approximately 20 000 lines of code) and apparently difficult to maintain – despite its popularity in WSN research community, in 2012 there is still no port of TinyDB to TinyOS 2.x (which itself was released in 2006).

TinyScript [Levis 2004a] is a high-level WSN language that compiles to Maté virtual machine bytecode. It uses event-driven execution model and is dynamically typed (although explicit data conversion operators must be used). The language was one of the first to address the problem of high-level WSN application programming. Byte code generated by TinyScript compiler is usually very short compared to sensor node MCU machine code, allowing efficient run-time reprogramming of the motes. All source code in TinyScript applications must be written as part of predefined event handler, a concept that was shown to be confusing for novice users [Miller 2009].

There also is another approach to sensor network programming – using web services. One example of this approach is **TinySOA** web service architecture [Avilés-López 2009]. The architecture allows programmers to "access" sensor networks by using their preferred programming language. The architecture was evaluated in a usability test and is used for a real world application – crop monitoring in precision agriculture. The API is characterized as "simple", and since a custom programming language can be used to access it, IDE of programmer's choice can be used. The one large drawback of TinySOA is the very limited intra-network data processing support. All sensor data is sent to a gateway, using network-wide measurement interval.

The authors of **WASP** [Bai 2009] categorize all WSN applications into seven archetypes, and present WASP, a simple language that is targeted towards one of the archetypes. The authors advocate design of archetype-specific languages in general. Although the idea is not without its merits, we feel that designing a universal WSN programming language is a more promising approach. It may very well be that a particular domain expert only has to work within one WSN archetype; however, not being knowledgeable in WSN, she will have difficulties recognizing the proper archetype and therefore also the language best suited for her needs. Furthermore, if the application requirements (and also the archetype) changes, she will be required to study and use a new language, rather than a different subset of the same language. Finally, a single, general-purpose WSN programming language is much more likely to gain the "critical mass" of popularity required to reach domain experts outside the small WSN programming community, and even in the community itself. [Bai 2009] is notable as one of the few papers that has performed a scientific user study. The authors compared WASP with four other languages and got favorable results. The results were even improved in the second version of the test, where a modified version WASP2 was used. The chief difference between WASP and WASP2 is the presence of IDE: another point typically ignored by other researchers.

1.6 Concluding remarks

The hardware of sensor networks is instrumental in determining the non-functional requirements of sensor network software. The heavy resource constraints of sensor network hardware stem mostly from the fact that it is powered from low capacity energy sources that frequently are non-rechargeable. However, there is also another factor – more constrained hardware is cheaper to build. As preliminary market research data (Table 1.1) shows, the code memory and RAM capacity of sensor network microcontrollers have relatively *low* correlation with its energy efficiency, but *high* correlation with its price. Therefore, developing more resource-efficient software (with smaller code memory and RAM footprints) allows to significantly reduce hardware costs.

Many of the prospective sensor network application developers are novice programmers. They are not likely to have extensive academic background in computer programming. Therefore, general purpose programming languages are not well suited for them; instead, DSL are more promising tools for sensor network application development for this target audience, because they allow to keep the notation minimal. Many of the issues that novice programmers face are related explicitly to the nature of imperative programming, therefore it makes sense to use language that is declarative in the core. Application examples or code templates should be easily accessible, as they are helpful for novices, who lack skills for program composition on their own.

The plethora of existing wireless sensor network programming approaches cover the software design space quite well, and each has its own strengths. However, each also has its own problems. Many are unsuitable for novice programmers. The few approaches that are targeted specifically to novice programmers could be improved if declarative approach was used [Miller 2009] or generality of the approach was increased [Bai 2009]. Macroprogramming (for example, [Kothari 2007], [Gummadi 2005] and [Newton 2007]) and query processing (for example, [Madden 2005] and [Wood 2008]) approaches have limited generality and sometimes are too abstract for the target audience of domain experts. The approaches that use abstractions that are familiar to the system programmer [Cao 2008], fail to convince us that they are well-suited for sensor networks. The approaches that adapt existing programming languages [Sivieri 2012] [pys 2013] are heavyweight; furthermore, they are not as well suited for novices as specifically-developed languages (DSL) can be. The commercial approaches are not very popular and also require extensive hardware resources (Squawk [Simon 2006] on Sun SPOT mote).

Chapter 2

Building system-level support

Contents

2.1 Introduction

A wireless sensor network operating system should provide clean and flexible services that allow the developer to create low duty-cycle applications naturally, and handle highly concurrent execution flow reliably (Section 1.4.1). TinyOS [Hill 2000] was the first to address this challenge. It is a small, efficient and highly influential system created at the beginning of 2000-ties at the University of Berkeley. Unfortunately, the system has gained notoriety as being difficult to learn [Levis 2012]. Sensor network operating systems developed after TinyOS tried to increase usability in several ways: by allowing to write applications in plain C [Dunkels 2004], using preemptive multithreading [Bhatti 2005], or supporting UNIX-like abstractions [Cao 2008].

As the sensor network research field matures, there is an increasing need for an operating system targeted towards applications rather than research itself. MansOS [Elsts 2012e] is a WSN OS developed with this goal in mind. It is implemented in plain C; one of the objectives of MansOS since its inception has been declared "to be easy-to-use for people without extensive WSN research background".

We believe that while TinyOS did great by introducing many new ideas suitable for WSN, more attention could have been devoted making these ideas accessible for broader range of programmers. The ideas in TinyOS could have more real-world impact and be better understood if they were made available for plain C programmers.

The objective of MansOS related to the state of art is to complement TinyOS in the areas of usability and learnability, and to complement other C-based WSN operating systems (Contiki, Mantis, LiteOS) with more superior technical solutions in some areas.

The author of this dissertation is not the sole author of MansOS operating system. Notably, the initial architecture of MansOS, initial example applications and initial platforms (*telosb* and *pc* out of the 10 current) were developed by Leo Selavo, who also had the initial vision and outlined initial design directions. The initial multithreading scheduler (replaced in 2011) was ported from Mantis OS by Girts Strazdins. Girts was also responsible for other aspects of MansOS, including ports to Atmega and Epic mote (now defunct) platforms, unified implementation of SPI protocol, port of protothreads from Contiki, and drivers for several devices, including the internal ADC module and the external flash chip. Andrey Vihrov contributed a file-system, device drivers, and several ideas related to function-granularity optimizations in application build process. The MansOS IDE and a few drivers were developed by Janis Judvaitis. Reinholds Zviedris contributed a port to MSP430 Launchpad platform. The web interface was initially developed by the author and greatly improved by Dzintars Kanasevics and Georgijs Kanonirs. Several others have contributed device drivers, bugfixes, bug reports and general suggestions. However, most of the aspects of MansOS analyzed in this thesis (including the component selection mechanism, the current multithreading scheduler, thread safety mechanisms, accurate time accounting, four-level code architecture, reprogramming mechanism) are developed by the author himself.

This chapter uses materials from [Elsts 2012e] and [Elsts 2013b]. Section about MansOS applications also uses text from [Zviedris 2010] and [Elsts 2014].

2.2 Code architecture

2.2.1 Motivation and overview

The portability of an software artifact is dependent on the amount of platform-specific code it has; there is an inverse correlation between these two variables. Since one of our design goals is to minimize the effort required to port MansOS to new hardware platforms, the code architecture is designed to reduce platform-specific code proportion. The design ideas author has developed (together with Girts Strazdins) and adopted, include: separate platform-specific and architecture-specific code; write multiplatform drivers and system code, where possible; allow the application user to access hardware drivers at all levels, but also provide hardware-independent API.

The hardware abstraction model in MansOS (Fig. 2.1) is based on a key observation from [Handziski 2005]: due to requirements of energy efficiency in WSN it is not enough to expose only a single, strictly platform-independent hardware abstraction layer. The users should be allowed to exploit device-specific hardware features for increased efficiency and flexibility.

In MansOS, chip-specific code is separated from platform-specific and platform-independent routines. Driver code is designed to be platform-independent where possible, therefore a single MansOS driver may be usable across multiple platforms. Examples of such multi-platform drivers include Texas Instruments CC2420 radio chip driver, Sensirion SHT humidity and temperature sensor driver, and Maxim Integrated DS2401 serial chip driver.

Initially MansOS adopted a three-level code organization. Hardware chip drivers were in the lowest layer, platform-specific code in the middle layer, and hardware-independent code above them. However, this model had two drawbacks:

- First, with introduction of support for multiple similar hardware platforms (for example, the MSP430 MCU based family: TelosB, Epic mote, and “generic” MSP430) most of platform-specific code was actually repeated in the source directories of all platforms. As a result, code maintainability rapidly declined, because every modification had to be repeated in several directories.
- Second, implementation was not clearly separated from interface. While this was acceptable for lower-level components such as hardware chip drivers, which essentially were self-contained device drivers and were placed each in its own subdirectory, it was not so acceptable for the upper-layer interface, because this interface was meant to be used by people who are not experts in

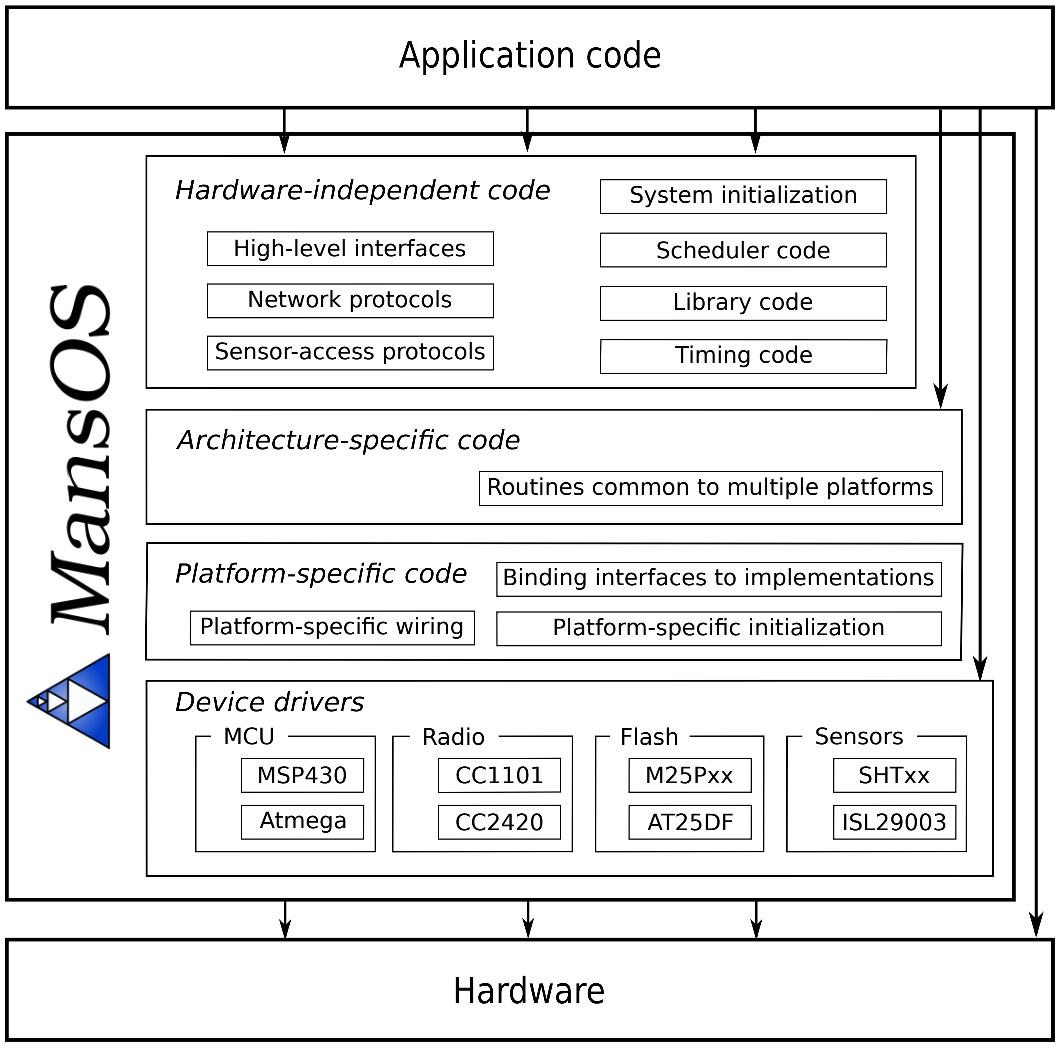


Figure 2.1: Architecture layers of MansOS code

the internals of MansOS. It is neither neccessary nor desirable to show them the details of the implementation.

To solve the latter issue, high-level API was separated from the rest of the code and put in `include` directory. Nevertheless, the advanced user is still allowed to use platform-specific and chip-specific code for utmost efficiency and platform-specific features, as platform and chip-specific header files are included in the top-level include file.

The code repetition issue was solved by separating architecture and platform layers. To take a concrete example, before the separation TelosB platform directory had 695 lines of code (evaluated excluding comments and empty lines); a lot of this code was similar or equal to code in similar platforms, for example, to the code in the directory of Epic Mote hardware platform. After the change, TelosB plaform directory had only 216 lines of code. There are a few other places where platform-specific code has always been present in MansOS (for example, in the `chips` directory, and in the `Makefile` directory), but the size of this code is not large. At the moment, MansOS supports ten hardware platforms, but only three hardware (MCU) architectures: Texas Instruments MSP430, Atmel AVR, and x86. Adding support for a TelosB-like sensor device such as AdvancticSYS XM1000 [AdvanticSYS 2013] now became possible within a few hours.

Taking into account these changes, the final MansOS code architecture consists of four layers:

- device-specific code (placed in directory `chips`) – drivers for individual devices and microcontrollers;
- architecture-specific code (directory `arch`) – code particular to a specific architecture (such as MSP430 or AVR);
- platform-specific code (directory `platforms`) – code particular to a specific platform (such as TelosB, Zolertia Z1, or Arduino).
- platform independent code. This code is further separated in two parts: the *interface* (directory `include`) and the *implementation* (directory `hil`, named after "hardware interface layer").

Architecture-specific code and platform-specific code include the so-called "wiring" (i.e. matching specific software interfaces with corresponding hardware interfaces), function binding (i.e. matching specific software interfaces with their software implementations), and definitions of platform or architecture-specific constants and algorithms. To take an example, radio driver's interface is defined in the hardware interface layer. At compile time the interface is bound to a specific implementation, which is chosen at the `platform` layer, containing the glue code. For TelosB platform, CC2420 radio driver is chosen, and so on. The driver code (at chip layer) implement hardware access functionality.

2.2.2 The proportion of device-dependent code

The term portability is used to refer to "the ability of a software unit to be ported (to a given environment). A program is portable if and to the degree that the cost of porting is less than the cost of redevelopment." [Mooney 1993]. Our objective is to support multiple hardware platforms, and (with a lower priority) to support multiple hardware architectures. Clearly, a system with no hardware-specific code would have the perfect portability; unfortunately, it is not possible in practice. In converse, a system that has only code that is specific to a single hardware platform is completely nonportable.

As a working approximation we assume that MansOS code is hardware-specific (device-dependent) in the case when it is platform- or architecture-specific, and also in the case when it is chip-specific (i.e. situated in `platform`, `architecture` and `chip` directories). The error in this assumption is likely to cause the results of the following evaluations to be skewed towards fewer lines of portable code detected

Table 2.1: Source code organization. The table lists directory names in MansOS, and their most closely corresponding counterparts in TinyOS, Contiki and Mantis

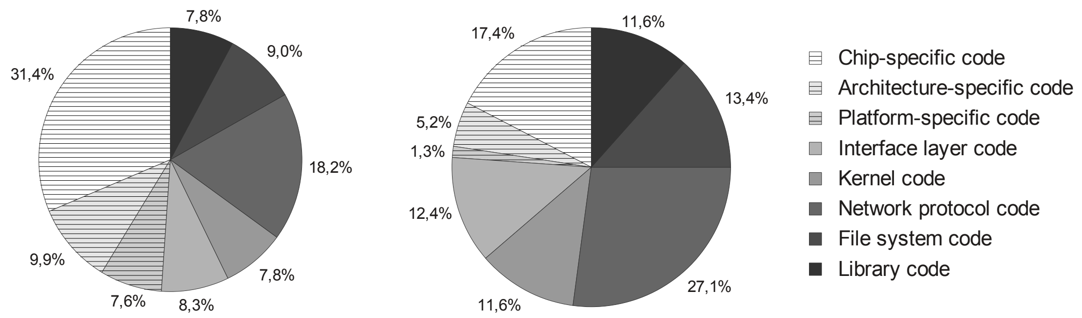
MansOS	TinyOS	Contiki	Mantis
`platforms`	`platforms`	`platform`	`kernel`
`arch`	`platforms` `chips`	`platform` `cpu`	`kernel`

`chips`	`chips`	`cpu`
		`core/dev`
		`platform/<name>/dev`
			`dev`
			`com`

`include`	`interfaces`	`core`	`<directory>/include`
`hil`	`interfaces`	`core`	`sys`
`kernel`	`system`	`core/sys`	`kernel`
`lib`	`lib`	`core/lib`	`lib`

than it is actually the case – as mentioned above, some of MansOS chip drivers are cross-platform, therefore also device-independent to some extent. However, we go by the rule of thumb that different sensor network hardware platforms use distinct set of chips, and that therefore chip drivers are device-dependent code.

The code that is device-dependent must be redeveloped when MansOS is ported to a new hardware architecture; the code that device-independent may be reused. Therefore, the portability of MansOS can be approximated by finding out the proportion of device-independent code.



(a) The proportion of each component group's code, all platforms

(b) Same, TelosB platform

Figure 2.2: MansOS code analysis. The area filled with horizontal line pattern corresponds to device-specific code

Fig. 2.2 shows the breakdown of MansOS code with regard to components. In total there was 25 701 non-empty non-comment lines in the core MansOS on 18th May, 2013 (version control system revision 918). While approximately half of MansOS code is platform dependent (48.9 %, Fig. 2.2a), comparison is more fair when a specific platform is fixed, because the amount of hardware-dependent code varies greatly with the number of hardware platforms supported. When TelosB is selected as the platform (Fig. 2.2b), only a quarter of the code (23.9 %) turns out to be chip- or platform-specific. Most of the hardware-dependent code is plain C; assembly language is used only in a few, specific places, such as thread context switching (in contrast to LiteOS: see Section 1.4).

Also, the proportion of platform-specific code is small when compared to the rest of device-dependent code. This means that the system should be easy to port to platforms that have the same hardware architecture as TelosB, and similar sets of chips. Indeed, that was the case when porting to AdvancticSYS XM1000 (see Section 2.2.1 and Section 2.2.3).

2.2.3 MansOS code architecture compared with the state of art

MansOS code organization has both similarities and differences to TinyOS, Contiki and Mantis (Table 2.1). In general, the amount of platform-specific code is lower in MansOS compared to the other OS because of three reasons:

- *architecture*-specific code is separated from *platform*-specific code (partially also done in Contiki);
- code for system initialization is unified across all platforms (partially also done in Mantis);
- device drivers and system services (e.g. the scheduler) are implemented in platform-independent way where possible (partially also done in all of the other OS).

Architecture-specific code roughly corresponds to to `cpu` directory in Contiki, but the unification of shared code is not a complete in Contiki as it is in MansOS. For example, only in MansOS and Mantis the periodic timer interrupt handler code (the “heartbeat” of the system) is unified and shared by all platforms.

The core system initialization code is unified for all platforms in MansOS. Platform-specific initialization code is still present, but significantly shorter in comparison: the system initialization file (`kernelmain.c`) contains 206 lines of code, while a platform’s initialization file has only 11.3 lines of code on average! This unification was made possible because of the component selection mechanism (Section 2.3). The platform-specific conmponents are enabled in platform’s configuration file; therefore, component initialization code can be put in kernel code; initialization code of a particular components either is or isn’t called depending on whether the selected platform has this component. The scheduler code is also unified for all platforms; only relatively small platform-specific fragments such as macros for context switching are put in platform directories.

Device drivers are portable because they use software abstractions for common tasks, such as SPI bus access, rather than talk to hardware directly. The tradeoff of this design choice is reduced hardware access speed is some cases. However, high data bandwidth support is usually not among the primary requirements of WSN applications [Strazdins 2013].

For a case study of a specific hardware platform we are going to take AdvancticSYS XM1000 sensor device. The official Contiki code (evaluated including Makefiles, excluding comments and empty lines) for XM1000 sensor device provided by AdvancticSYS has 1920 lines of code (excluding BSL script, application examples, and file `checkpoint-arch.c` because MansOS has no equivalent functionality). Their TinyOS code has 1228 lines of code (excluding BSL script, MAC protocol, and CC2420X driver). MansOS, in turn, has only 629 lines of XM1000 specific code. The rest of the code is reused from TelosB platform without copying it. Some of the new code (for MSP430 Series-2 MCU support) is actually reused by other platforms (e.g. Zolertia Z1). Excluding that code, there are only 251 lines, in other words, 7.6 times less than in Contiki, and 4.9 times less than in TinyOS. Clearly, this leads to faster portability, and, more importantly, better maintainability.

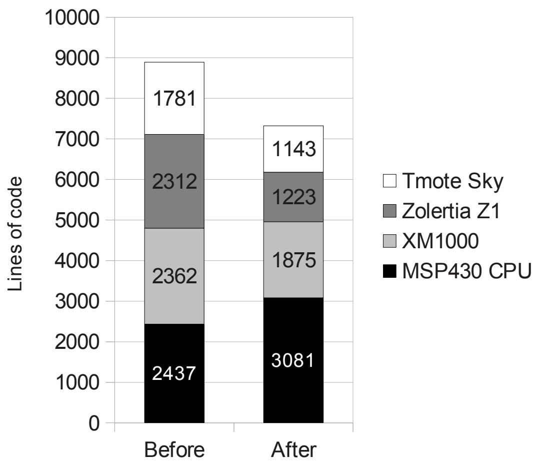


Figure 2.3: The result of applying MansOS code organization to three Contiki hardware platforms and MSP430 MCU architecture

In order to show that the gains are present specifically because of MansOS code organization and not due to other unrelated reasons, we also modified the source code of Contiki itself. We selected just three hardware platforms (Tmote Sky, Zolertia Z1, and AdvancticSYS XM1000) that share a common MCU architecture MSP430 (in contrast, they do not share a common MCU family, as the examples include both MSP430 Series-1 and MSP430 Series-2 MCU!). We simply moved all files that correspond to `arch` code in MansOS from `platform` folders into `cpu/msp430` folder in Contiki. Creating a new `arch` folder in Contiki would have lead to better code organization, but was not necessary to prove the point of this optimization. The result (excluding platform-specific example application code) is shown in Fig. 2.3. The Contiki source code size was reduced by 1570 lines of code, which is 17.7 % of code in directories of these three platforms (`platform/sky`, `platform/z1`, `platform/xm1000`) and `cpu/msp430` directory, in sum. Applying the same technique to more platforms with the same MCU architecture would lead to increased gains. On the other hand, supporting more MCU architectures would lead to unchanged code sizes. Since the code is simply moved around rather than written anew, the net change in line count cannot be positive.

Another difference between MansOS and Contiki source code organization is related to function binding: in Contiki it is not done at compile time. Compile-time binding allows to reduce both binary code size and RAM usage, as well as run-time overhead. (This idea is not specific to MansOS; compile-time binding is also preferred in design of TinyOS and Mantis.) To take a concrete example, in Contiki the radio driver is accessed through function pointers in `struct radio_driver` structure. The structure itself takes twenty bytes in RAM. Furthermore, indirect function calls have to be used, which adds two byte flash usage overhead for each call, as well as CPU run-time overhead, because an extra `mov` instruction is generated. The extra `mov` takes two CPU cycles to execute, because on MSP430 instruction execution takes an extra CPU cycle for each memory access. In MansOS all calls are direct, therefore extra resources are not used.

The authors of Mantis adopt another approach to fight code duplication by separating code that is common to all microcontroller platforms, in contrast to code that is used for the x86 platform. However, architecture specific-code separation (as in MansOS, and, partially, in Contiki) is more scalable; the Mantis approach is going to feasible only while the number microcontroller architectures is small and the microcontrollers are reasonably similar.

More interestingly, for peripheral access *device* abstraction is extensively used. Each device driver can be accessed using POSIX-like system call API, consisting of six functions: `dev_read(dev, buf, count)` ("reads count bytes from a device and stores the result in the provided buffer"[1]), `dev_write(dev, buf, count)` ("writes count bytes to a device from the provided buffer"), `dev_mode(dev, mode)` ("set the mode of a device"), `dev_ioctl(dev, request, args...)` (used to "send arbitrary commands to a device"), `dev_open(dev)` ("gain an exclusive lock on a device driver"), and `dev_close(dev)` ("release an exclusive lock on a device driver"). For communication, a similar (but separate) API exists, including functions: `com_send(iface, buf)` ("send data over specified interface"), `com_recv(iface)` ("receive data from a specified interface"), `com_mode(iface, mode)` ("pass the mode setting down to the given interface"), and `com_ioctl(iface, request, args...)` ("pass ioctl flag and data down to the given interface").

Listing 2.1: Flash memory access example in Mantis

```
uint32_t address = 0x1234;
uint32_t writeValue = 123, readValue;

dev_mode(DEV_TELOS_FLASH, DEV_MODE_ON);
dev_ioctl(DEV_TELOS_FLASH, TELOS_FLASH_SECT_ERASE, address);
dev_ioctl(DEV_TELOS_FLASH, DEV_SEEK, address);
dev_write(DEV_TELOS_FLASH, &writeValue, sizeof(writeValue));
dev_ioctl(DEV_TELOS_FLASH, DEV_SEEK, address);
```

[1]Descriptions here and further are quoted from Mantis source code

```
    dev_read(DEV_TELOS_FLASH, &readValue, sizeof(readValue));
    dev_ioctl(DEV_TELOS_FLASH, TELOS_FLASH_BULK_ERASE);
```

Listing 2.2: Equivalent code in MansOS

```
    uint32_t address = 0x1234;
    uint32_t writeValue = 123, readValue;

    extFlashWake();
    extFlashEraseSector(address);
    extFlashWrite(address, &writeValue, sizeof(writeValue));
    extFlashRead(address, &readValue, sizeof(readValue));
    extFlashBulkErase();
```

In our opinion, the device abstraction, at least as implemented in Mantis, is a tradeoff that is not worth the cost. It is much more useful in classical UNIX systems, where the overhead of C language switch-case statements can be easily ignored, but the system call interface must be kept small because it crosses the boundary between user and kernel spaces. In sensor network hardware, there is no explicit system/user boundary, and using ordinary calls-by-name allow to reduce flash-memory usage significantly. Last but not least, using several distinct functions with descriptive names and parameter sets compared to calling `dev_ioctl` repeatedly leads to code that has fewer redundant details. For example, Mantis code in Listing 2.1 can be replaced with MansOS code in Listing 2.2.

2.3 The component selection mechanism

Since a sensor network operating system usually runs only one application, it is possible to optimize the OS in application-specific way. In practice this means that only a subset of MansOS source-level components is selected for compilation, linking and inclusion in the final binary image that contains both application code and OS code (Fig. 2.4). A mechanism that allows the user to select OS components in a natural way should be integrated in the application development process.

TinyOS allows (and requires) the application developer to specify which components to use. This must be done in application-specific source file: the *configuration file*. This leads to two benefits: firstly, optimal binary code size, as only components that are actually used are included in the final application; secondly, extensibility and flexibility, as it now becomes simple to integrate platform-specific or application-specific components in the OS. The configuration file in TinyOS is processed by the nesC compiler.

The solution that MansOS adopts is similar to TinyOS: to use an additional, “configuration” file next to application source code. The configuration file allows to select or exclude components to use in a specific application. It is also useful for setting system-wide policies, e.g. the radio channel to use, the serial port baudrate to use, the MAC protocol to use, its queue size and so on. These policies cannot be set in C source code in equally efficient way, because setting them either in a function call or by changing a global variable adds to run-time overhead.

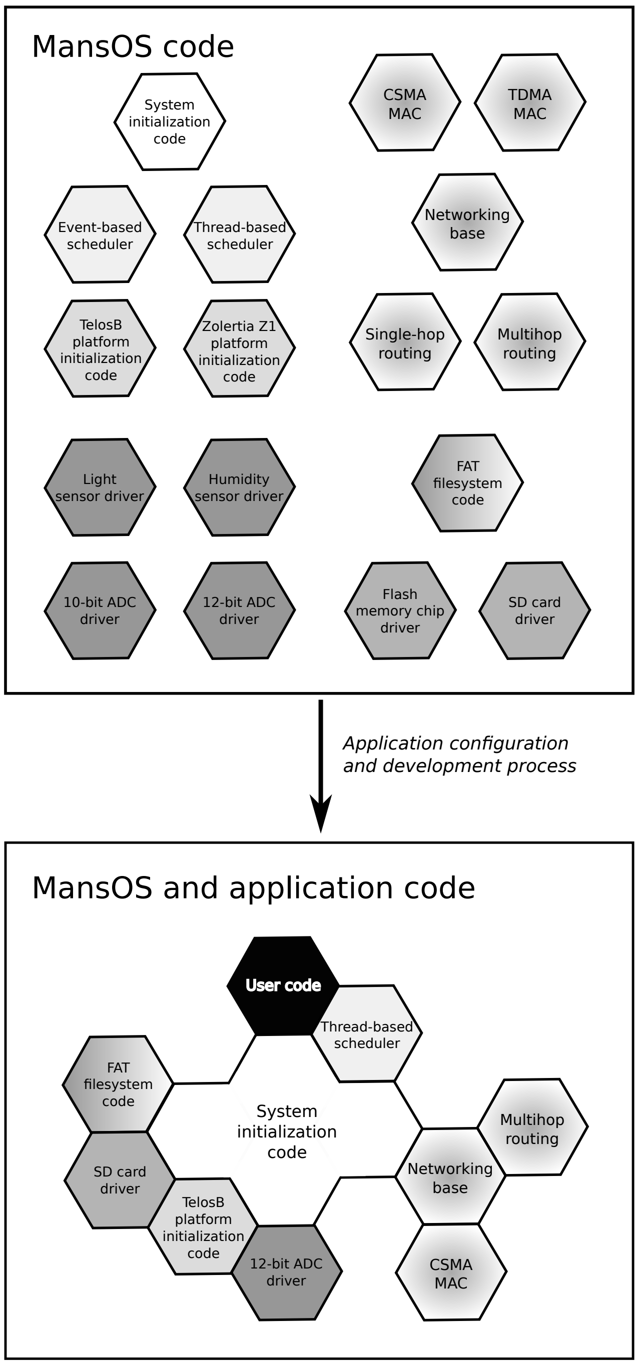


Figure 2.4: In the application development and configuration process, user code is added and a subset of MansOS components is selected and configured

Even though these policies could be set in system header files at compile time, the MansOS approach is better, because by requiring that the configuration is put in a distinct file, MansOS enforces clear separation, at the level of source code, between *what* is used and *how* it is used. Furthermore, now there is no need to modify *system* header files in order to get *application* specific behavior.

In contrast, other C-based operating systems such as Contiki and Mantis have limited means to achieve something similar. Mantis allows to select components with large granularity – high-level components are compiled as a separate libraries, which may or may not be linked against the application. Unfortunately, not only this becomes unreasonable if smaller component granularity is needed, but also tends to create unresolvable circular dependencies between the libraries.

As mentioned in Section 1.4.4, Contiki adopts the "most of functionality is included by default" strategy, which leads to large code size and suboptimal energy usage. However, there is a set of options that can to enabled and configured at compile time by using "project's configuration file" (a C header). This Contiki approach has severe limitations:

- It is not uniform, but rather ad-hoc for each of the few configurable components.
- It applies only to the networking stack rather than to the whole system.
- It cannot be used to exclude or include files in a build; at least not without stretching the definition of "reconfiguration" as far as including `Makefile` changes in it, which means that a distinct file written in another language has to be modified by the user.
- It is not enabled by default. Furthermore, it cannot be enabled by default, at least not in an easy way, because of technical limitations. There is no way how to instruct C compiler to "#include a file only if it exists" at the level of C source code. As a consequence, in order to use some non-default configuration in Contiki, first a new header file has to be created (called `project-conf.h`), then compilation flags have to be modified in `Makefile` in order to enable preprocessing of this header. Therefore usability is decreased, especially for inexperienced users.

Even when "minimalistic" configuration is selected, the resulting application's resource usage is much larger in Contiki than in other operating systems (Fig. 2.22).

The syntax of configuration file is very simple; it consists of variable assignments (Listing 2.3). There are three types of variables:

- Variables with prefix "`USE_`". Used to include or exclude components from the final executable.
- Variables with prefix "`CONST_`". Used to define compile-time constants.
- Special variables, like "`DEBUG`". Used for various purposes as described in the documentation of MansOS[1].

[1]`http://mansos.edi.lv/wiki/doku.php?id=mansos_configuration_system`

Listing 2.3: Example MansOS configuration file

```
#
# Configuration file of a multihop routing testing application
#

# Use threaded kernel
USE_THREADS=y
# Use nondefault radio channel
CONST_RADIO_CHANNEL=11
# Use SAD routing and MAC protocols
CONST_MAC_PROTOCOL=MAC_PROTOCOL_SAD
CONST_ROUTING_PROTOCOL=ROUTING_PROTOCOL_SAD
# Function as data forwarder in the network (other roles are possible)
USE_ROLE_FORWARDER=y
# Enable debugging
DEBUG=y
```

The syntax is compatible with GNU make utility, allowing simple integration in MansOS build process; in contrast to TinyOS (with its nesC compiler), no additional compiler has to be installed. Learning requirements also are reduced compared with TinyOS, as MansOS configuration file allows to write nothing more than variable assignments, but nesC syntax includes more details (see Section 3.5.1).

The resulting component granularity is larger than in TinyOS. For example, there are no separate components for each timer, but rather one component that allows to use timer functionality as such. Similarly, there are no separate components for reading from external flash memory (`BlockRead` in TinyOS) and for writing to it (`BlockWrite` in TinyOS), but rather a single component for external flash memory access (with read/write/erase functions). This allows to keep the configuration file small while remaining reasonably flexible.

MansOS build system ensures that each platform by default enables a reasonable set of components. Dependencies between components may exist; they are resolved automatically. Therefore, the component selection in MansOS is a semi-automatic. The novice user may successfully start using MansOS without being aware that such a mechanism exists. On the other hand, the experienced user may optimize or extend her applications easily by excluding and including components manually.

Essentially, the component selection mechanism allows to move towards the vision of sensor network software outlined in [Romer 2004]: “a modular software architecture would then be needed, together with tools that would semi-automatically select the implementations that best fitted the application and hardware requirements”.

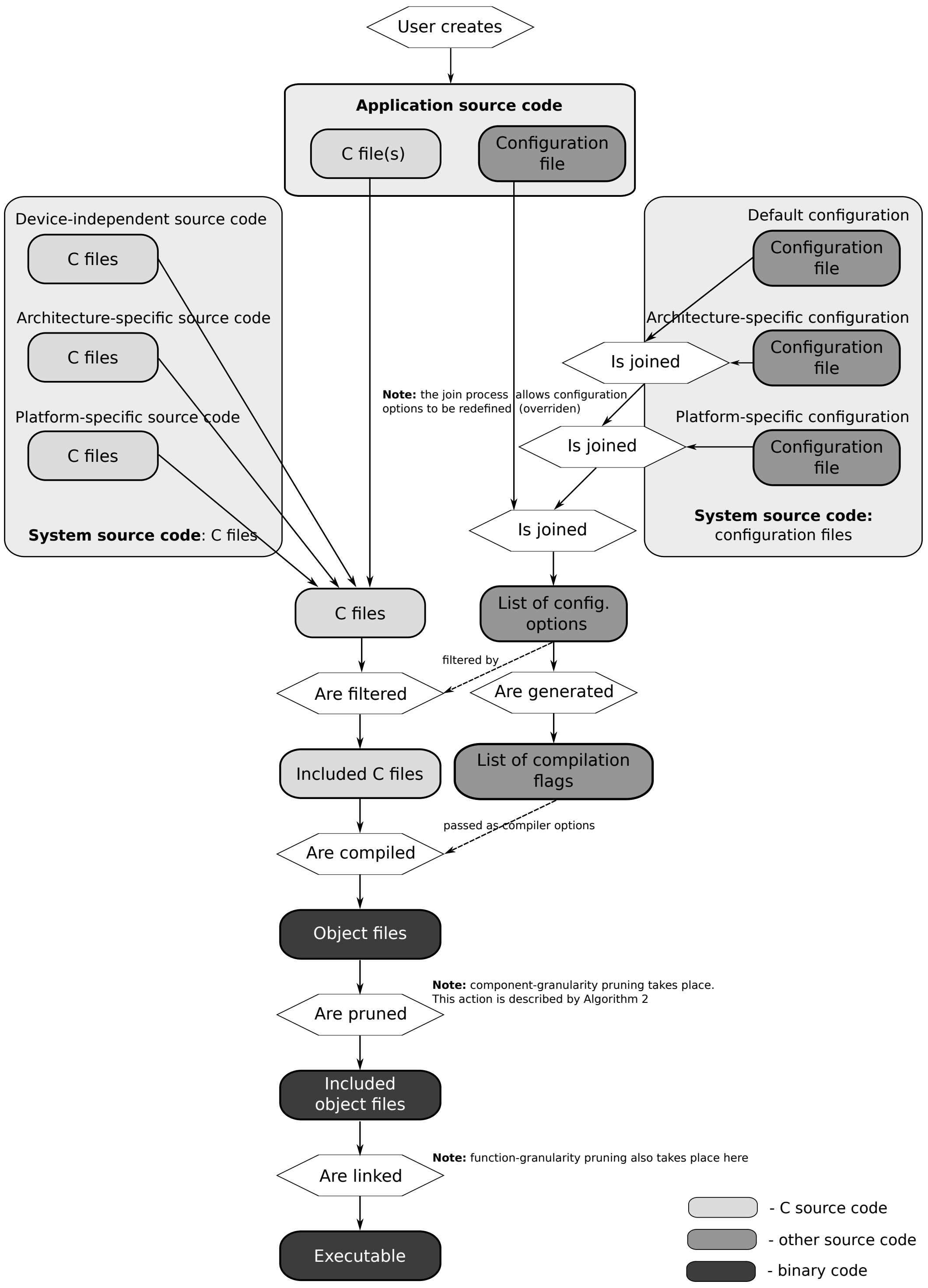

User creates
Application source code
C file(s)
Configuration file
Device-independent source code
C files
Architecture-specific source code
C files
Platform-specific source code
C files
System source code: C files
Default configuration
Configuration file
Architecture-specific configuration
Configuration file
Platform-specific configuration
Configuration file
System source code: configuration files
Is joined
Is joined
Is joined
Note: the join process allows configuration options to be redefined (overriden)
C files
List of config. options
filtered by
Are filtered
Are generated
Included C files
List of compilation flags
passed as compiler options
Are compiled
Object files
Note: component-granularity pruning takes place. This action is described by Algorithm 2
Are pruned
Included object files
Note: function-granularity pruning also takes place here
Are linked
Executable
- C source code
- other source code
- binary code

Figure 2.5: The build process of a MansOS application

There are three benefits brought by such a mechanism:

- Modularity: components that implement the same interface can be used interchangeably, often without changing application's C source code. For example, two versions of the kernel (event-based and thread-based) can be replaced with each another by changing just a single line in the configuration file.
- Extensibility: application-specific or platform-specific components can be easily added and used. For example, if a platform has a specific actuator and a driver of that actuator is added to MansOS, the actuator can be enabled for that platform by adding only a line in *platform's* configuration file. If the hardware actuator is also added for a specific application on another platform, it can be enabled by adding a line in *application's* configuration file.
- Efficiency: unused components are not included in the binary image (Algorithm 2.2). This allows to achieve more efficient binary code and RAM usage, which in turn reduces prototyping and debugging time (Section 2.6.2).

The whole build process is shown in Fig. 2.5 and elaborated in more technical detail in Algorithm 2.1.

The pseudo-code of Algorithm 2.2 is also given (it corresponds to the "Are pruned" step in Fig. 2.5). This algorithm takes list of object files and list of configuration options as inputs, and produces the list list of object files that may have some code that should be included in the final binary image of the executable. It implements file-granularity binary code component selection, therefore allows to optimize the size of the binary image.

In addition to this, MansOS (similarly to Contiki) also allows to achieve function-granularity binary code selection, using combination of GCC compiler and linker features. If the source code is compiled with `-fdata-sections -ffunction-sections` flags, each function is put in a separate object file section. If the code is then linked with `-gc-sections` option, the linker discards unused sections. Since each `.text` subsection contains just a single function, the result is the desired.

Now the question may arise – why there is a need for *file-granularity* binary code component selection (Algorithm 2.2) when the finer-grained *function-granularity* selection is already in place? Still, code size measurements confirm that the best results are achieved when both file-granularity and function-granularity garbage collection are combined (Section 2.6.2, Fig. 2.20). The answer lies in the fact that the selection mechanism is semi-automatic. In order to provide reasonable defaults for the unexperienced user, a few components (such as ADC driver) are enabled by default. If they are present in the final binary image, they also must be initialized by the kernel of the operating system itself. For each enabled component, an initialization function is called from the system-wide initialization code placed in the kernel. Because of this, a reference to component's initialization function is added to the kernel. Now the component cannot be garbage-collected automatically, as the linker will see the component as being used. However, if the component is not

Algorithm 2.1 The component selection algorithm

```
Include the default configuration file
Include platform-specific configuration file                ▷ Overrides default settings
Include application-specific configuration file, if present ▷ Overrides platform settings
                                                ▷ Build compile-time flags and definitions
for each variable defined in form USE_xxx=y do
    Add compile-time definition USE_xxx=1
end for
for each variable defined in form CONST_xxx=value do
    Add compile-time definition xxx=value
end for
                                                ▷ Select potentially required files
$applicationSourceFiles \leftarrow$ application's .c files
$systemSourceFiles \leftarrow$hardware-independent $\cup$ architecture- $\cup$ platform-specific .c files
$enabledSourceFiles \leftarrow \varnothing$
for $file \in allSourceFiles$ do
    $component \leftarrow$ the component this $file$ belongs to
    if USE_$component = y$ then
        $enabledSourceFiles \leftarrow enabledSourceFiles \cup \{file\}$
    end if
end for
                                                ▷ Compile all potentially required files
$applicationObjectFiles \leftarrow \textsc{compile}(applicationSourceFiles)$
$systemObjectFiles \leftarrow \textsc{compile}(systemSourceFiles)$

                                                ▷ Link the required files together
$executable \leftarrow \textsc{link}(\textsc{FindRequiredFiles}(applicationObjectFiles, systemObjectFiles))$
                       ▷ The function FindRequiredFiles is defined in Algorithm 2.2
```

Algorithm 2.2 Pruning of unused components

```
function FindRequiredFiles($appFiles, systemFiles$)
    $librarySymbols \leftarrow$ {'printf', 'malloc', ...}    ▷ All functions from C library
    $symbolsFound \leftarrow librarySymbols$
    $symbolsUnresolved \leftarrow \varnothing$
                              ▷ Determine which symbols are required by the application
    for $file \in appFiles$ do
        $localExports \leftarrow$ symbols exported from $file$
        $localImports \leftarrow$ external symbols required to $file$
        $symbolsUnresolved \leftarrow symbolsUnresolved \cup (localImports \setminus symbolsFound)$
        $symbolsUnresolved \leftarrow symbolsUnresolved \setminus localExports$
        $symbolsFound \leftarrow symbolsFound \cup localExports$
    end for
                    ▷ Determine which files are required; start with all application's files
    $result \leftarrow appFiles$
                    ▷ Process system files while all symbols are found or all files included
    $progress \leftarrow true$
    while $symbolsUnresolved \neq \varnothing$ AND $systemFiles \neq \varnothing$ AND $progress$ do
        $progress \leftarrow false$
        for $file \in systemFiles$ do
            $localExports \leftarrow$ symbols exported from $file$
            $localImports \leftarrow$ external symbols required to $file$
                                                  ▷ Does this file include new symbols?
            if $symbolsUnresolved \cap localExports = \varnothing$ then continue
            end if
            $symbolsUnresolved \leftarrow symbolsUnresolved \setminus localExports$
            $systemFiles \leftarrow systemFiles \setminus \{file\}$
            $symbolsUnresolved \leftarrow symbolsUnresolved \cup (localImports \setminus symbolsFound)$
            $symbolsFound \leftarrow symbolsFound \cup localExports$
            $result \leftarrow result \cup \{file\}$                    ▷ This file will be linked
            $progress \leftarrow true$
        end for
    end while
    return result
end function
```

used, neither from user's application code nor by other components, but only from the system-wide initialization code, we know that it is safe to exclude the component from the binary image. Algorithm 2.2 does exactly that: excludes components that have these properties. However, after exclusion there is a new problem: how to deal with the reference to the initialization function? This problem is solved by the GCC extension `__attribute__((weak))`, which is used to signal that the GCC linker may leave the reference to the function unresolved in the final binary image.

In practice, there is another aspect why file-granularity optimization may be needed: for several versions of *msp430-gcc* linker, the function-granularity optimization was working incorrectly [Elsts 2012e] and had to be disabled[1]. Also, build systems other than the GCC may not implement this optimization at all.

In order to show that MansOS approach effectively scales to other sensor network operating systems, we partially implemented the component selection algorithm not only for MansOS itself, but also for Contiki. The implementation contains just Algorithm 2.1, and does not include Algorithm 2.2. It consists of a patch that is applied to the development version of the Contiki source code repository, changes 37 source files and includes 272 code insertions and 41 deletions[2]. The results are promising and are discussed in the evaluation section of this chapter (Section 2.6.2).

2.4 Execution models

One example of an interchangeable component in MansOS is the kernel itself. Two versions of the core scheduler exists: one that uses preemptive threads, and one that uses only callback functions that react on specific events.

2.4.1 Threads or events?

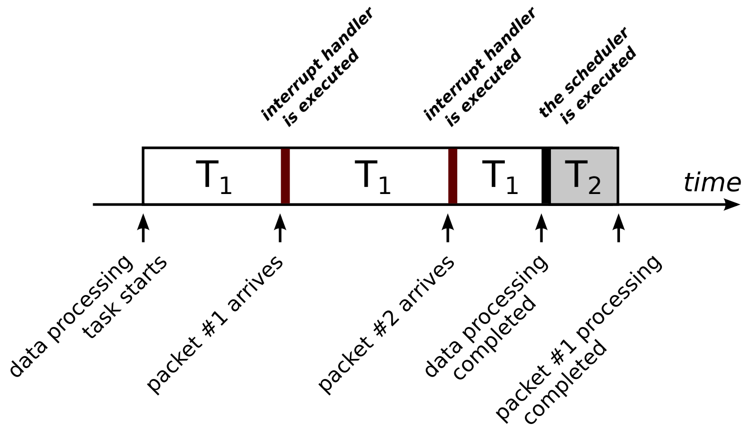
Much has been written as part of the debate about thread-based versus event-based execution models. Lauer and Needham in their seminal article [Lauer 1979] attempt to settle the debate by proving that both models are equivalent from theoretical point of view. However, that does not make discussion about the two options meaningless, merely constrain it to practical implications; the pragmatic difference remains. For example, the thread approach requires more memory (for additional stack space). On the other hand, the event approach uses run-to-completion event handlers that may make applications unresponsive. Usability of the two models also is not the same. Since the Lauer and Needham's paper, new arguments have been formulated by both sides of the debate [Von Behren 2003] [Ousterhout 1996].

When narrowing the debate to WSN OS context, the outcome is still inconclusive, and none of the approaches is superior in all contexts [Duffy 2008]. Threads

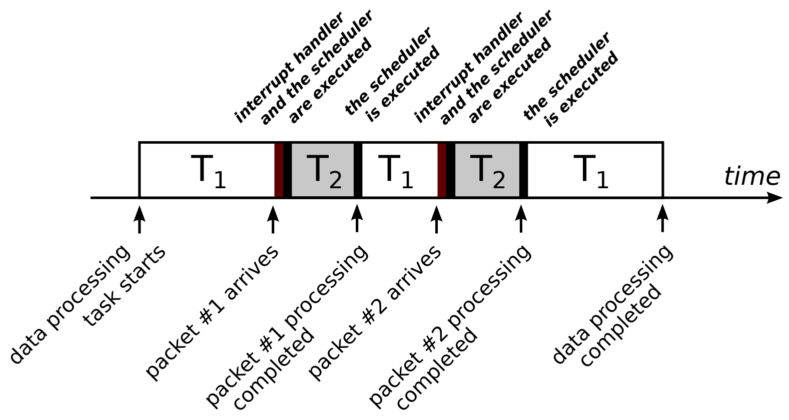
[1]For detailed description of the problem, see: `http://comments.gmane.org/gmane.comp.hardware.texas-instruments.msp430.gcc.user/9615`

[2]The modified version of the Contiki operating system is available at `https://github.com/atiselsts/contiki-optimized`.

are generally seen as easier-to use, because it is believed that threaded code better reflects the control flow of applications (in WSN context). On the other hand, event-based code is not only more efficient, but also better reflects the underlying hardware execution model (which is always event-driven).


interrupt handler is executed
interrupt handler is executed
the scheduler is executed
T1
T1
T1
T2
time
data processing task starts
packet #1 arrives
packet #2 arrives
data processing completed
packet #1 processing completed

(a) Run-to-completion scheduling


interrupt handler and the scheduler are executed
the scheduler is executed
interrupt handler and the scheduler are executed
the scheduler is executed
T1
T2
T1
T2
T1
time
data processing task starts
packet #1 arrives
packet #1 processing completed
packet #2 arrives
packet #2 processing completed
data processing completed

(b) Preemptive scheduling

Figure 2.6: Scheduling algorithm behavior: run-to-completion scheduling causes the second packet to be dropped

To show one of the problems with run-to-completion scheduling (as implemented by TinyOS) in WSN context let us consider a specific example (Fig. 2.6), an application that includes two kinds of tasks:

- A data processing task T_1 that takes long to complete. For example, T_1 may execute a compression algorithm or Fourier transform on sensor data.
- A packet forwarding task T_2 that requires low delay. The T_2 does not take long to complete, but has another limitation: whenever there is a data packet in a temporary buffer waiting for T_2 to start execution, each subsequently received data packet is dropped. (In WSN systems single-packet-only buffers are common.)

Fig. 2.6a illustrates event-based approach to scheduling. First, the T_1 task starts and runs to completion. Somewhere in the middle of its execution, a network packet (packet #1) is received; this event is signaled by the hardware, and processed immediately by the corresponding interrupt handler. The handler reads the packet and puts it in a temporary buffer. Some time later, another packet (packet #2) is received and another interrupt processed. Since the temporary buffer is already full, the packet is dropped. After T_1 is completed, T_2 finally starts and processes the first packet.

Fig. 2.6b illustrates thread-based approach. Task T_1 starts, but can be preempted (interrupted) by task T_2. As soon as the first packet is received, the scheduler starts task T_2, which processes the packet and frees the temporary buffer. Afterwards, T_1 is continued from where it left. The same sequence is repeated once the second packet arrives. No packets are lost: in this example run-to-completion scheduling causes the second packet to be dropped, while preemptive scheduling does not. (The second packet would be lost only if it arrived before T_2 was completed.) The example also shows that the preemptive multithreading approach allows to easier implement more strict QoS requirements in the network; the delay in packet processing is much smaller in Fig. 2.6b than in Fig. 2.6a, with no additional effort from the programmer. Alternative would be to partition the long-running task T_1 in smaller sub-tasks "by hand". However, it may require substantial programmer effort [Bhatti 2005] – first, the WSN programmer may be unfamiliar with the semantics and internal details of the algorithm, especially if the algorithm is ported to a WSN system from some common library, rather than developed by the programmer or his colleagues; second, some algorithms may be hard to divide in parts by design (e.g. compression algorithms).

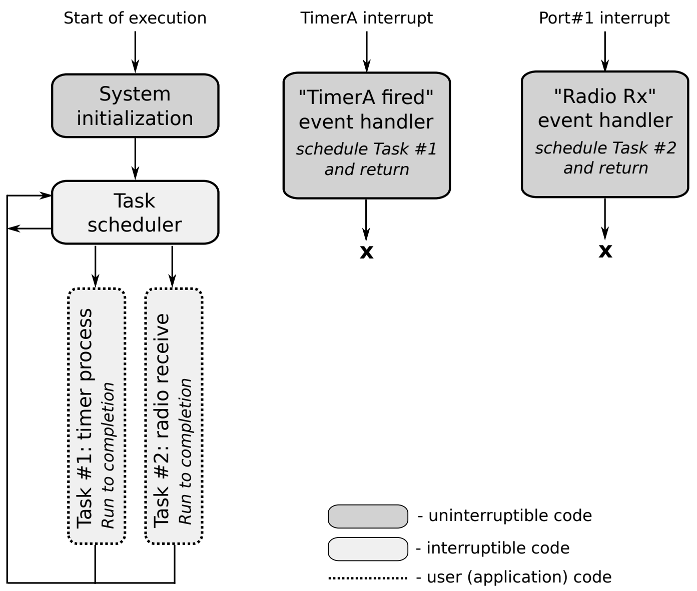


Figure 2.7: State-of-art event-based scheduler (TinyOS)

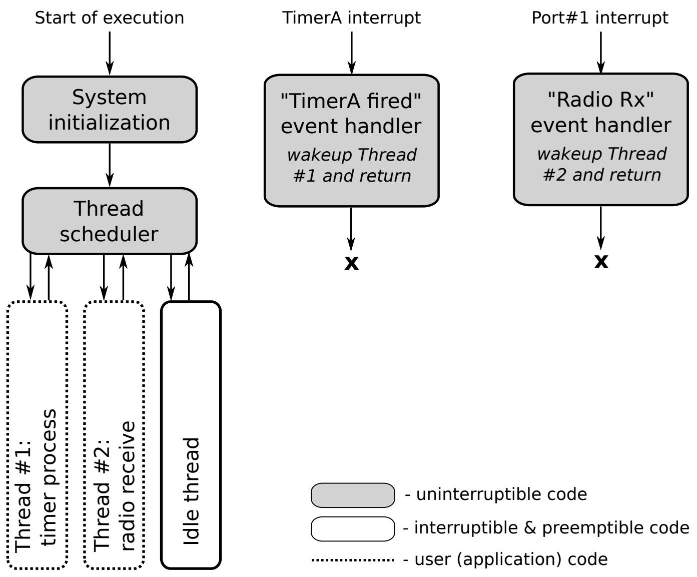


Figure 2.8: State-of-art preemptive multithreading scheduler (Mantis)

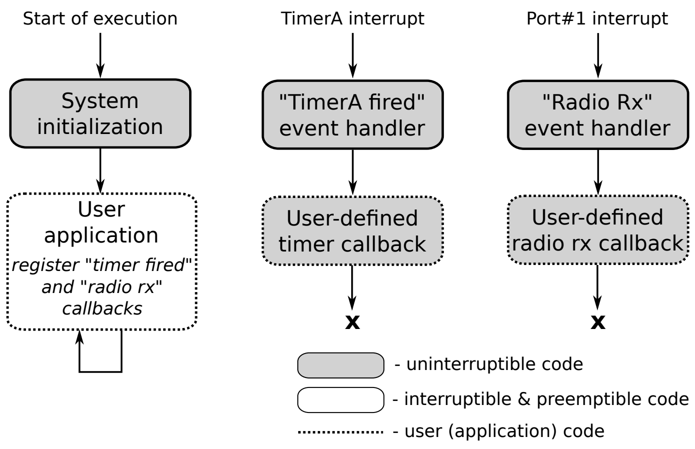

Start of execution
TimerA interrupt
Port#1 interrupt
System initialization
"TimerA fired" event handler
"Radio Rx" event handler
User application
register "timer fired" and "radio rx" callbacks
User-defined timer callback
User-defined radio rx callback
x
x
- uninterruptible code
- interruptible & preemptible code
- user (application) code

Figure 2.9: MansOS event-based scheduler

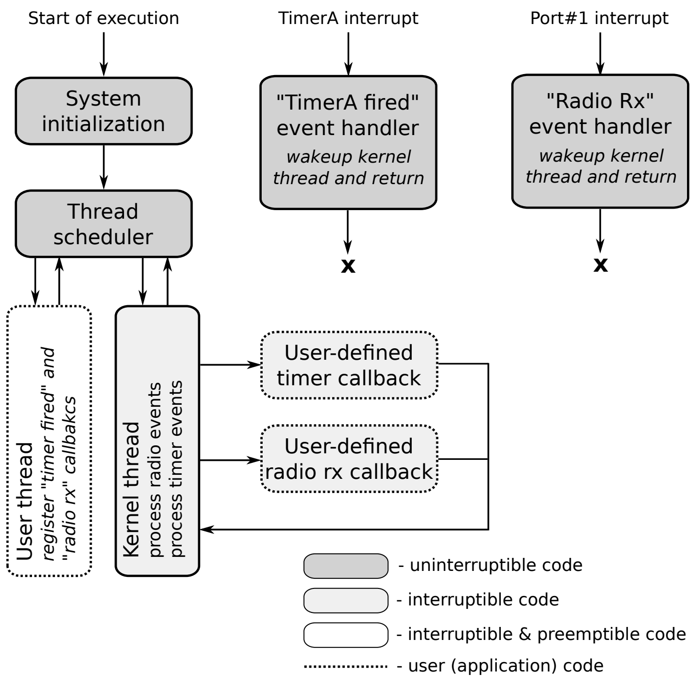

Start of execution
TimerA interrupt
Port#1 interrupt
System initialization
"TimerA fired" event handler
wakeup kernel thread and return
"Radio Rx" event handler
wakeup kernel thread and return
Thread scheduler
x
x
User thread
register "timer fired" and "radio rx" callbakcs
Kernel thread
process radio events
process timer events
User-defined timer callback
User-defined radio rx callback
- uninterruptible code
- interruptible code
- interruptible & preemptible code
- user (application) code

Figure 2.10: MansOS multithreading scheduler

Fig. 2.7 to Fig. 2.10 depicts execution flow of a simple application that reacts on radio packets and timer events. The different approaches to scheduling in TinyOS, Mantis, and MansOS are visualized.

Taking into account the conclusion from the discussion above (each model has its own benefits), it is no surprise that existing WSN OS already fully cover the design space, with systems like TinyOS 1 being restricted to events (Fig. 2.7), TinyOS 2 allowing to optionally enable TOSThreads [Klues 2009], and Mantis or LiteOS using preemptive multithreading (Fig. 2.8). All of this hints that none of the two options is superlative (superior in all contexts). Automated methods for conversion from threaded applications to event-based applications do exist, for an example see [McCartney 2011], but they are thread implementation-specific.

A different approach is taken by Contiki, which uses so-called *protothreads*, which is a cooperative stack-less execution model. This approach is not without minor usability problems: lack of thread-local variables, nontransparent execution flow because a lot of macros are used, and nontrivial learning curve because of a novel programming paradigm. However, protothreads can be easily incorporated in WSN OS as a library running on top of either event-based or thread-based kernel, therefore it is not necessary to discuss them as a full-scale alternative to the two other models. (They are not depicted a separate figure in this thesis; if taken as implemented in Contiki, they would look like the multithreading picture (Fig. 2.8), except that there would be no "idle" thread, and the other threads would not be preemptible. A protothread can be preempted only by event handlers; however, it can also give up the MCU to another protothread voluntarily.)

MansOS implements both event-based (Fig. 2.9) and thread-based (Fig. 2.10) kernels, and allows the user to choose (at compile time) which to use. The event based version's core functionality revolves around processing a global list of software timers, on which callback functions for various tasks are chained. The multithreaded version adopts the same idea as TOSThreads and uses at least two priority levels: the highest level for the kernel thread and lower levels for user threads. Therefore the kernel thread, which handles delay-sensitive tasks like packet forwarding, can never be preempted by a user thread. The specific contribution of MansOS is simplification of the scheduler made taking into account WSN specifics: the low number of threads expected, and separation between kernel and user threads.

2.4.1.1 Event-based execution

This (Fig. 2.9) is the default implementation used in MansOS. In event-based execution model, the user registers callbacks and writes code for callback handler functions.

Take software timers (named *alarms* in MansOS) as an example. Alarm callback function pointers are put in a global list, ordered by alarm firing time. The list is processed in the periodic timer interrupt handler, executed 1000 times per second. Therefore, software timers with precision up to 1 ms are available. (Section 2.5.2 has more in-depth discussion about timer precision and accuracy; however, for this example such an approximation is sufficient.)

Similar callbacks can be registered for packet reception, whether serial or radio. User callbacks, if present, are executed immediately after their corresponding interrupt handler is entered (when the hardware signals arrival of new data), therefore the delay is as small as possible. However, user callback code is executed in the interrupt context and can cause problems either if the execution blocks for too long, or if the user code re-enables interrupts. In the first case, the result is a completely blocked system. In the second case, nested interrupts become possible, so the developer has to ensure that all of OS code is reentrant. Also, stack overflow becomes a very real threat. An active interrupt handler creates a new stack frame in the stack space of the previous execution context; N nested, active interrupt handlers means that N new stack frames are allocated. Since the value of N cannot be determined at compile time, it is impossible to reserve enough stack space for all cases.

To enter a low power mode, the user has to manually call `sleep()` function in the main loop of her application.

2.4.1.2 Threaded execution

Thread implementation in a WSN OS can be simplified if two observations are taken into account. First, the number of threads typically required by a WSN application is small (Section 1.4.4.4). If the operating system provides core functionality as separate services, a single user thread that allows blocking function calls is sufficient for most of simple WSN applications. Second, in contrast to processes in a desktop OS, threads in WSN OS can be expected to be cooperative (because they all belong to the same application). The first observation motivates the OS to provide simpler scheduler version by default, supporting only two threads. The second allows to think about fairness guarantees as an optional, rather than critical requirement. Implementation that takes into account these observations leads to reduced code memory and RAM usage (see the evaluation in Section 2.6.3).

In MansOS, at least two threads are always created: a user thread and the kernel thread. Multiple user threads are optionally available. In the latter case, two scheduling policies are available: *round-robin*, in which the least-recently run user thread is always selected, and *priority-based*, in which the thread with the highest priority is always selected from threads that are ready to run (not in sleeping or waiting states). The round-robin policy is active by default, the priority-based is selectable by a configuration option.

A thread can be in one of three states: *ready* to run, *sleeping* until wakeup time or interrupt, and *waiting* when blocked by some other thread, or waiting for an external condition.

The MansOS scheduling pseudocode is given in Algorithm 2.3. The pseudocode is for the round-robin scheduler; priority based version simply selects the thread that has the highest priority from those that are not in waiting or sleeping states.

We note that the algorithms at the moment do not incorporate deadlock detection and prevention (e.g. in case of priority inversion). This is an area for future work. In any case, deadlock detection would not be as simple as checking whether all threads are in `WAITING` state, as that is a valid case – the threads may be waiting for some external event signaled by a hardware interrupt. Also, even though a `WAITING` thread may be "selected" by the scheduler, the thread will not become active; the thread will check for the condition it was waiting for, determine that it is still not fulfilled, and give back control over execution to the scheduler (i.e. call `schedule()`).

Both the policy and the maximal number of threads are selected at compile-time. This allows to include only one of the functions `SelectNextThreadSimple` and `SelectNextThreadManyThreads` in the binary image. Specifically, binary image size

Algorithm 2.3 Thread scheduling algorithm (for round-robin scheduling)

```
function Schedule( )
                                        ▷ currentThread is the currenly active thread
    Save all MCU registers to stack
    if $numberOfThreads = 2$ then
        $nextThread \leftarrow$ SelectNextThreadSimple( )
    else
        $nextThread \leftarrow$ SelectNextThreadManyThreads( )
    end if
    $currentThread.stackPointer \leftarrow$ stack point register value
    $currentThread \leftarrow nextThread$
    Set stack pointer register value from $currentThread.stackPointer$
    Restore all MCU registers from stack
end function

function SelectNextThreadSimple( )      ▷ Next thread's selection, 2 threads total
                                        ▷ otherThread is the currenly inactive thread
    if $otherThread.state = READY$ then return $otherThread$
    end if
    if $currentThread.state = WAITING$ then return $otherThread$
    end if
    if $otherThread.state = WAITING$ then return $currentThread$
    end if
    if $currentThread.timeUntilWakeup > otherThread.timeUntilWakeup$ then
        return $otherThread$
    end if
    return $currentThread$
end function
                                        ▷ Next thread's selection, > 2 threads total
function SelectNextThreadManyThreads( )
    From all threads with $state = READY$, return thread run least recently
    From all threads with $state = SLEEPING$, return thread with the soonest wakeup
    return $currentThread$
end function
```

is significantly reduced when only a single user thread is used (see Fig. 2.24 and the discussion surrounding it). The decision not to use thread queues or other advanced data structures was motivated by the suboptimal code size and performance of the Mantis scheduler in the context of a typical WSN application (Section 2.6.3).

Mutexes are available as means of synchronization. Sequential execution of two threads can be implemented using a mutex.

2.4.2 Avoiding stack overflow

Use of multiple threads may make application code easier to write, but care must be taken because it may also make it easier to write *incorrect* application code. Specifically, stack overflow becomes a very real and hard-to-debug problem if small and fixed-size thread stacks are used. To facilitate writing more reliable programs, MansOS includes a tool for worst-case stack usage analysis. The analysis process takes short time (fractions of second) to complete and is integrated in the MansOS build process. If the reported worst-case stack usage is larger than the stack allocated, the build is aborted with a descriptive error message.

Since the use of function pointers is not prohibited in MansOS, the analysis cannot always be accurate. However, by specifying certain policies to be followed by the core system developers, we limit this problem. Namely, in the core system we do not use recursion, and, in case of thread-based kernel, do not use calls-by-pointer in interrupt handlers. Therefore, the analysis is accurate for *user* threads unless the user herself uses recursion or calls-by-pointer in application code. In the latter cases, warning messages are issued: “Recursion detected” and “Unhandled call-by-pointer in function ...”, respectively, followed by the message: “Stack usage analysis results may be incorrect!”.

It is hard to avoid calls-by-pointer in the *kernel* thread completely. However, estimating the stack usage for the kernel thread is not as important (on a typical system), since its stack is allocated by the compiler and, if sufficient system memory is available, can grow to significant size without overlapping any other memory regions. (With a “typical system” we mean TelosB or its clones, where flash memory is the limiting resource instead of RAM. [Levis 2012] mentions a rule of thumb that a typical TinyOS WSN application uses approximately 10 times more flash memory than RAM memory; we have observed similar results for MansOS.)

In cases when compile-time stack usage estimation is insufficient, MansOS provides run-time stack overflow early-detection macros. They include a `STACK_GUARD` macro, which causes immediate abort of the system in case stack pointer is out of the currently used stack region. This macro can be used from application's code, but is also called from several places in MansOS system source code, being included in a number of functions that are usually at the bottom of the call chain.

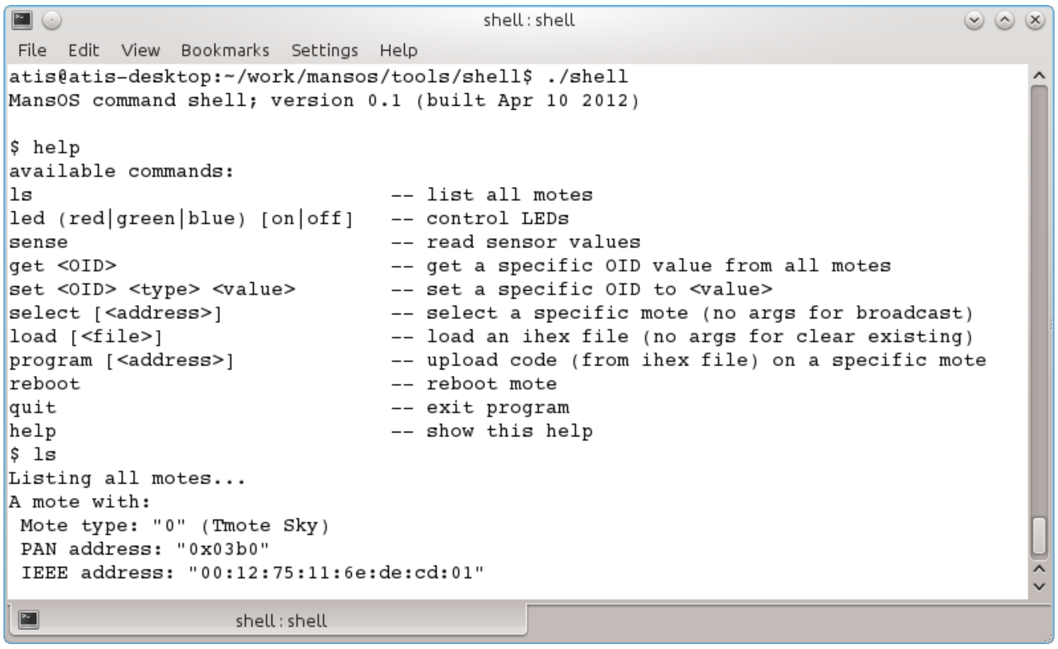
2.5 Interfaces and functionality

2.5.1 Run-time management and reprogramming

Occasionally WSN applications require interactive management of specific resources, whether sensors, actuators, or software variables. Management interface is also a useful tool to non-programmers; it gives them a hands-on experience with the network.

2.5.1.1 Management via shell

In order to perform run-time management an efficient protocol (named *SSMP*) is implemented. Each available resource on the sensor device is assigned an object ID, which uniquely identifies the resource in device-local scope. An object ID is a binary string with arbitrary length. The object IDs form a hierarchical space; for example, object ID for LEDs is a prefix for object ID for red LED. In this way, multiple resources can be accessed at once, by specifying only the common prefix of their object IDs. To continue with the example, not only each LED, such as the red LED may be turned on/off separately by completely specifying its n byte long object ID OID_{redLED} (which is “0x01, 0x03, 0x00” in MansOS), but also all LEDs can be turned on/off *at once* by specifying the $n-1$ byte long prefix of OID_{redLED} (“0x01, 0x03” in MansOS).



```
atis@atis-desktop:~/work/mansos/tools/shell$ ./shell
MansOS command shell; version 0.1 (built Apr 10 2012)

$ help
available commands:
ls                              -- list all motes
led (red|green|blue) [on|off]   -- control LEDs
sense                           -- read sensor values
get <OID>                       -- get a specific OID value from all motes
set <OID> <type> <value>        -- set a specific OID to <value>
select [<address>]              -- select a specific mote (no args for broadcast)
load [<file>]                   -- load an ihex file (no args for clear existing)
program [<address>]             -- upload code (from ihex file) on a specific mote
reboot                          -- reboot mote
quit                            -- exit program
help                            -- show this help
$ ls
Listing all motes...
A mote with:
 Mote type: "0" (Tmote Sky)
 PAN address: "0x03b0"
 IEEE address: "00:12:75:11:6e:de:cd:01"
```

Figure 2.11: MansOS shell interface

A command line shell is implemented to provide user access to the management interface (Fig. 2.11). The shell can be used both interactively, and non-interactively (from scripts). The shell allows to access arbitrary object IDs. Named shortcuts for the some common object IDs are present, for example `led` (LED control and status) and `sense` (sensor status). The shell can function in broadcast or unicast modes; in the first, all reachable sensor devices in the network are accessed; in the second, only a single device, specified by its address, is queried.

2.5.1.2 Management via web interface

Later on, a different management interface was added (Fig. 2.12). This interface is usable from web browser, and allows to specify sensor read intervals by entering numbers in fields in a simple web form. The form is generated in platform-specific way, so only the sensors that are present are shown. The interface also features a form that visualizes sensor data.

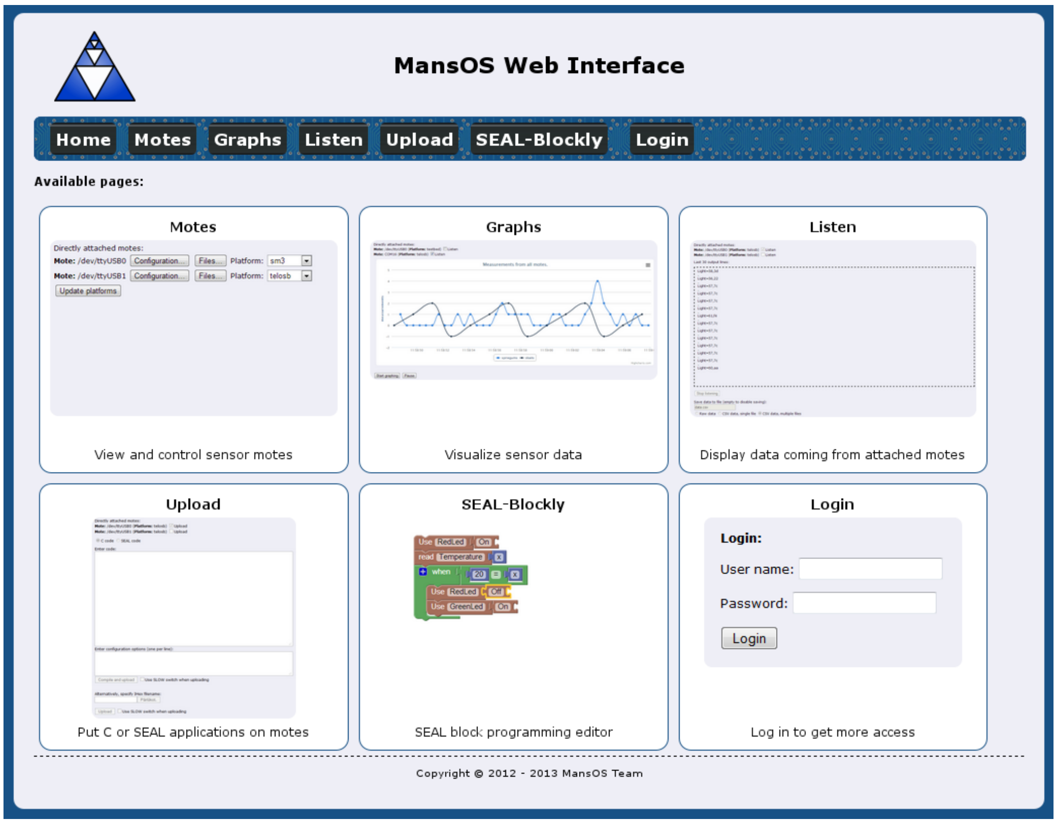

MansOS Web Interface
Home
Motes
Graphs
Listen
Upload
SEAL-Blockly
Login
Available pages:
Motes
View and control sensor motes
Graphs
Visualize sensor data
Listen
Display data coming from attached motes
Upload
Put C or SEAL applications on motes
SEAL-Blockly
SEAL block programming editor
Login
Login:
User name:
Password:
Login
Log in to get more access
Copyright © 2012 - 2013 MansOS Team

Figure 2.12: MansOS web interface

Both management interfaces are intended to be used from a computer that has base station device directly attached. The attachment does not need to be physical – if the device has, for example, so called Ethernet extension shield that adds support for TCP/IP networking, it can be attached "virtually", over Ethernet [Clark 1997].

Web interface is currently tailored for the needs of sensor network deployment in a test bed infrastructure, and presupposes a wired, Ethernet connection to each sensor node in the network. Integrating the web interface in a network with wireless sensor network links is a future work item.

Another plan for future work: not only provide a data entry form where the user can input the desired sensor reading intervals, but also generate SEAL / SEAL-Blockly (Chapter 3) application code based on this input. This would add another level of abstraction on top of the framework described in this thesis: a level where the user is not a programmer anymore, but merely inputs data a web form in order to program or configured a sensor network. Similar approach has been described by [Van Deursen 1997].

2.5.1.3 Reprogramming

Reprogramming is an action that replaces the binary code that is executing on a sensor network device. Normally, a sensor node can be reprogrammed without the support of its operating system, using a personal computer physically attached to a programming board, which in turn has direct physical connection to the sensor device. (In some cases the programming board can avoided, e.g. when the sensor device has an USB interface. However, the direct physical connection is still required.)

A more enticing form of reprogramming is *run-time reprogramming*, also called over-the-air reprogramming. The benefits of run-time reprogramming are apparent. For example, in our environmental monitoring use case (Chapter 4), reprogramming just nine sensor nodes in outdoor conditions took more than two hours. Using over-the-air reprogramming would reduce time requirements by an order of magnitude.

The task of run-time reprogramming is the same as that of "regular" reprogramming: transfer a binary image of an application from developer's computer (or a server) to a specific sensor mote; once the transfer is completed, execute the image. The image of an application includes both user code and operating system code.

Run-time reprogramming in MansOS is performed in four steps (the first three are performed in parallel):

1. Code loading. The binary code of an application is read from a file on disk and sent out to the sensor network.
2. Dissemination. The code is transported through the network.
3. Code storing. The code is received and stored on the target devices.
4. Image replacement. The target devices reprogram themselves and run the received application.

The *first step* is performed by MansOS shell (Fig. 2.11). It should be executed from a personal computer with a sensor node attached (the so-called "base station node"). For reprogramming purposes the shell is extended with Intel IHEX file parser and multiple reprogramming related commands (such as "`load [<file>] - load an ihex file`" and "`program [<address>] - upload code (from ihex file) on a specific mote`"). The shell sends out the code in binary data packets; each packet includes a memory address relative to the start of the executable, and a blob of machine code instructions among other information.

For the *second step*, SSMP is reused. A special object ID signaling binary data is used for code packets. The base station node functions as a gateway between the sensor network and the personal computer. Since the management protocol runtime is a system service in MansOS, no specific user application has to be developed for the base station. Almost any application can be used for this purpose; all that is required is to explicitly enable SSMP in the application's configuration file.

The *third step* is also done by another system service of MansOS. This, "reprogramming", service is included in application's binary image by specifying `USE_REPROGRAMMING=y` in its configuration file. The service subscribes to the SSMP service for binary machine code packets. Once a packet is received, it is written in external flash memory at an address determined from contents of the received packet. Once the whole image of an application is received, it is marked (in the external flash memory) as complete: ready to be used.

MansOS bootloader is responsible for the *final step*, the most complicated one: the replacement of active application with the new one, received in previous steps. The bootloader is another MansOS component which can be optionally included in application's binary file. If present, the bootloader is executed before any other MansOS code. If the reprogramming component has signaled the need to replace the existing program image (by writing a Boolean flag to the nonvolatile memory of sensor node before reboot), the bootloader performs the actual rewrite: it replaces MCU program memory contents with a new OS image taken from the device's external flash memory.

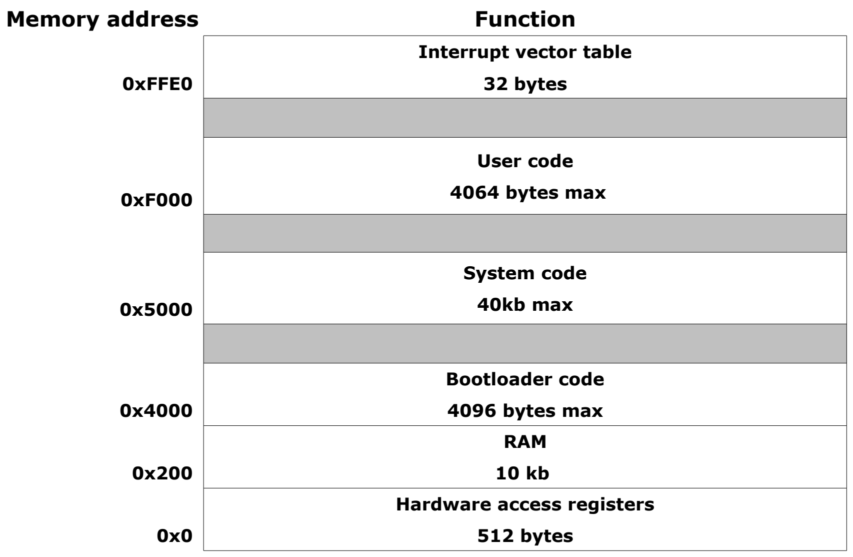


Figure 2.13: MansOS memory layout of a *Tmote Sky* application compiled with run-time reprogramming support

Partial run-time reprogramming is supported for energy-efficiency purposes. *Partial reprogramming* means that only portions of the binary image are replaced, rather than the whole image.

We observe that in WSN applications user code is smaller and require changes much more often than OS code. Therefore, it makes sense to separate system and user code, and allow to reprogram only one of them without changing the other. In MansOS, partial reprogramming is implemented through address space separation of system and user code (Fig. 2.13). For user section only 4 KB of memory space is allocated, since the user code can be expected to be much smaller, as evidenced later in Table 2.3. For an even more striking example, `DemoRedLed` reprogramming demo application in MansOS is 14259 bytes long, of which only 56 bytes are user code. Therefore, avoiding system code reprogramming leads to great efficiency gains.

However, the implementation of partial reprogramming in MansOS at the moment is conflicting with the component selection mechanism. Therefore it is not enabled by default and must be explicitly enabled in configuration file. Doing that disables both component-granularity and function-granularity optimizations. It is easy to see why: application-specific optimizations are not possible since the application logic itself may be subject to change!

2.5.2 Timing

When studying the source code state-of-art operating systems that run on TelosB platform, we made an interesting conclusion: none of the three systems analyzed (TinyOS, Mantis, and Contiki) provide high-accuracy millisecond-precision timers. Furthermore, out of the three only the latter provides high-accuracy long-term time accounting in SI units.

This is an usability problem because diversity of measurement units and lack of coordination between their naming in different areas may lead to confusion, and to occasional errors. The world, with few exceptions, has converged to the metric system (*Le Système international d'unités*) because using one, generally accepted system of measurements helps to avoid misunderstandings. In information technology, some confusion between binary and decimal units already exists (e.g. "kilobyte" is frequently used where "kibibyte" should be used); however, author's opinion is that there are few fields when such a confusion is as undesirable as the field of time measurements. Applications typically do not crash or have long-term problems because hardware memory capacity is 2.4 % (for example) smaller than expected; in contrast, timing errors are cumulative and potentially dangerous.

The cause of the problem lies in the fact that hardware timer ticks used for time accounting do not allow to have 1:1 mapping to milliseconds in whole numbers. In TelosB-based and similar platforms (Tmote Sky, Zolertia Z1 [Zolertia 2013], AdvanticSYS XM1000 [AdvanticSYS 2013], SADmote (Section 4.3), and others), two hardware clock sources are generally available: a high-frequency digital oscillator and a low-frequency crystal. If a timer is sourced from the digital oscillator, implementing a millisecond counter becomes trivial. Unfortunately, the oscillator is not stable enough for accurate long-term time accounting (for example, it has large temperature drift: 0.38 % per degree Celsius [MSP 2011]). Therefore the low-frequency crystal has to be used, as it is comparatively much more stable: for example, the standard crystal used in design of SADmote has only 20 parts-per-million maximal error, which corresponds approximately to clock drift of one minute per month. Since a hardware timer sourced from the crystal can have 32 768 Hz maximal granularity, it is not possible to trigger an interrupt *precisely every* millisecond.

While in active mode, Mantis is triggering the interrupt *approximately* every millisecond. In the interrupt function, plus one is added to the system time counter. In contrast, while in sleep mode, Mantis triggers the interrupt when the next event should be processed. In this case, a value calculated from the number of hardware timer ticks passed since last interrupt is added to a system time counter. Unfortunately, the timing error accumulates with time, and is nondeterministic (dependent on whether the system was in active or in sleep mode).

TinyOS puts the responsibility for time correction on the user. The accounted milliseconds are called "binary milliseconds" in TinyOS; every wall-clock second contains 1024 binary milliseconds. For novice users, this approach is confusing, as nothing in the name of `TMilli` suggests that it does not refer to the ordinary (SI time unit) milliseconds. The confusion is evidenced by the number of questions and help requests at TinyOS user mailing list[1].

The timing accuracy in the C code generated by TinyOS nesC compiler could be easily fixed: in fact, only one line must be changed, replacing a binary shift operation with multiplication. However, the TinyOS timers API [Sharp 2007] is so generic and multifaceted that introducing such a fix in it is much harder.

The Contiki solution is to not to provide milliseconds as time accounting units at all. Instead, `CLOCK_SECOND` macro is defined for mapping from hardware timer ticks to SI units. Contiki also provides a highly accurate global time counter, but only with second granularity.

The fact that such a timing inaccuracy (or lack of milliseconds as timing units as in Contiki) has existed for such a long time in WSN OS hints that one of the following might be the case:

a) accurate software-based time accounting is not required in WSN (either because it is not required at all, or can be achieved with other, such as hardware-based methods).
b) millisecond timers and accurate long-term time accounting in millisecond units is either not useful, or not feasible to implement on sensor network hardware.

In our experience we have met at least one case when (a) is false: such an accounting was required and could not be implemented using other means (at least not without rethinking the system architecture). In our precision agriculture deployment (Chapter 4) we met with a situation where the users of our system wanted to record timestamped microclimate data from multiple locations. Time-synchronization protocol was enabled in the network, but was of limited help: the

[1]For some examples, see: `http://mail.millennium.berkeley.edu/pipermail/tinyos-devel/2010-March/004317.html`, `http://mail.millennium.berkeley.edu/pipermail/tinyos-users/2003-October/000319.html`, and `https://www.millennium.berkeley.edu/pipermail/tinyos-help/2007-July/026965.html`

network often become partitioned, because some of intermediate nodes died, and because the users of the sensor system wanted us to deploy nodes even in places where network coverage was not planned. The sensor nodes we were using had neither Real Time Clock chip nor GNSS receiver (absence of both is typical for the current generation of WSN hardware [Strazdins 2013]). Therefore the only way to generate accurate timestamps was to rely on time accounting algorithms implemented in software. The case for software-based time accounting is also supported by the fact that both TinyOS and Contiki do provide an API for it; we criticize them for the limitations of this API, rater than for absence of it.

As for (b), while milliseconds as time units is not *required* by the user, it increases the usability of the operating system by allowing to use familiar rather than unfamiliar measurement units. One can say that there is need for the *millisecond abstraction* at the system level: i.e. the system should abstract away the application developer from the low-level details of hardware timing, unless using hardware explicity is enforced by application requirements. Additionally, if we want to build higher-level programming interfaces (such as SEAL: see Chapter 3) on top of the OS, they certainly will not use hardware ticks as time units. It is better if the underlying OS can provide 1:1 mapping to units used in higher-layer interfaces. In this case the mapping between SEAL code and C code is more clearly visible to the reader, and the generated code is more concise, as it contains no conversion functions or macros. (At the moment the generated C code is used not only by automated tools! We also have anecdotal evidence that people write SEAL applications and study the generated C code in order to learn sensor network programming in MansOS.)

Implementation alternatives require some thought. What we are looking for is a timing API that keeps track of the current time, and supports multiple software timers. There should be at least the following operations: scheduling, removing from schedule, and querying time-to-fire. Both the current-time counter and timer periods should be expressed in milliseconds.

The first option is to count hardware timer ticks at the system level, and convert timer milliseconds to hardware ticks every time a user exploits the timer API. This option can be discarded at once as having significant overhead due to a lot of division operations required.

The other option is to start with what Contiki has: a two-level hierarchy of accurate time counters, consisting of hardware tick-based units at the lower layer and a second counter at the higher layer. Developing the idea to match our requirements, the timer API may provide two versions of functions: one that takes parameters expressed as seconds, and another that takes milliseconds. Now only the latter version must use long-division for conversion, increasing the efficiency somehow. However, the API becomes less clear; usability is reduced.

Nevertheless, Contiki faces another performance penalty: to support timers with periods longer than hardware timer overflow interval, the system has to periodically wake up and modify all timers. This wastes MCU cycles, especially if the application uses lot of timers. The modification period may be increased by using lower time

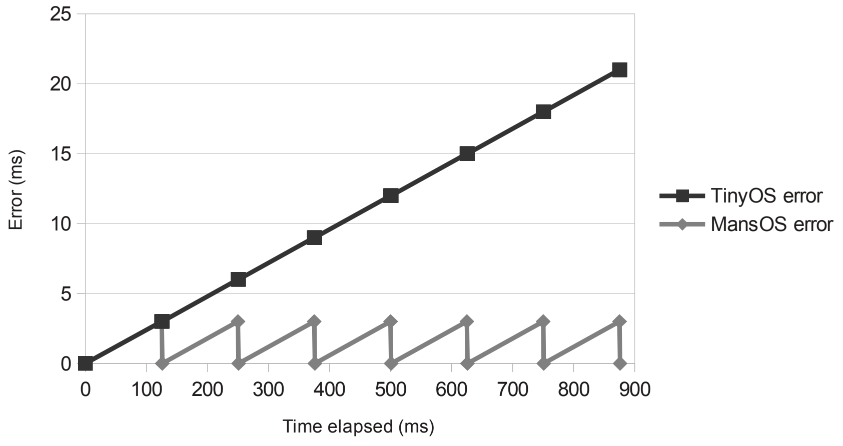


Figure 2.14: Timing accuracy comparison

unit granularity. However, in this case a third time accounting layer must be used to achieve high precision. (The reason why Contiki has `clock_fine()` function.) Additionally, in Contiki an interrupt is generated for each second-layer timer tick, even when the system is in low-power mode. This does not allow to achieve the smallest duty-cycle possible, and poses a problem for applications with very low power budget. (See also Section 2.6.5 for discussion about energy consumption of Contiki applications.)

Let us describe the capabilities of hardware in more detail. Once again, platforms similar to TelosB feature a high-accuracy, but low frequency crystal oscillator. The oscillator is accessible from software by using hardware timer API provided by the manufacturer of MSP430 MCU. The API consists of definitions that allow to work with maximum of two hardware timers: *TimerA* and *TimerB*. Each timer has multiple 16-bit *hardware counters* (not to be confused with the millisecond counter, which is a variable in code!). Their number is dependent on the model of the MCU; for example, for MSP430F1611 MCU (used in Tmote Sky) there are ten counters in total: three counters for TimerA (named TimerA0, TimerA1, and TimerA2) and seven counters for TimerB. In any case, MansOS wires the oscillator to the TimerA during system initialization sequence. Each counter can be independently configured to generate an interrupt once a custom numerical 16-bit value $T_{counter}$ value is reached. The frequency of the crystal oscillator is $F_{max} = 32\ 768\,\text{Hz}$.

Therefore the maximum interrupt generation frequency of all counters also is 32 768 Hz, which corresponds to period $T_{counter} = 1$. Table 2.2 shows other valid counter periods and corresponding interrupt frequencies. In contrast, interrupt frequencies such as 3, 10, 100 or 1000 Hz cannot be generated using counters with fixed period.

The objective is to develop algorithm that would be as simple as possible (because more complex algorithms require more code memory), would account milliseconds with reasonable accuracy in the short term, and would be error-free (have accuracy that is limited by hardware only) in the long term.

The idea adopted in MansOS is to use two hardware timers (see Algorithm 2.4):

Table 2.2: Examples of valid timer counter periods and corresponding interrupt frequencies

Period, $T_{counter}$	**Frequency, $F_{counter}$**
1 ticks	32 768 Hz
2 ticks	16 384 Hz
4 ticks	8192 Hz
128 ticks	256 Hz
256 ticks	128 Hz
2^{15} ticks	1 Hz
$2^{16} - 1$ ticks	~0.5 Hz

a "millisecond" timer running with *approximately* 1000 Hz frequency, and a "correction" timer that would correct the timer drift once it is large enough. A suitable value for millisecond timer is $T_{millisecond} = 32$, leading to $F_{millisecond} = 1024$. An ideal value for $F_{correction}$ in this case is 24 Hz. However, this ideal value does not correspond to a valid (i.e. integer) period. The largest factor of 24 that can be implemented with the counter is 8, therefore we set $F_{correction} = 8$ and $T_{correction} = 4096$. Every time the millisecond timer fires, the global millisecond time counter is increased by one. Every time the correction timer fires, the counter is decreased by three (because $24/8 = 3$). The accuracy error, which is cumulative in TinyOS, is kept bounded to 3 milliseconds in MansOS (Fig. 2.14).

Algorithm 2.4 The essential elements of MansOS time accounting algorithm

```
function Msp430InitClocks                         ▷ Called during system initialization
    $milliseconds \leftarrow 0$
    wire TimerA to the low-frequency crystal      ▷ 32 768 ticks per second
    set TimerA0 period to 32 ticks                ▷ 1024 times per second
    set TimerA1 period to 4096 ticks              ▷ 8 times per second
    start TimerA
end function

function TimerA0Interrupt                         ▷ Called on TimerA0 hardware interrupt
    $milliseconds \leftarrow milliseconds + 1$
end function

function TimerA1Interrupt                         ▷ Called on TimerA1 hardware interrupt
    $milliseconds \leftarrow milliseconds - 3$
end function
```

In order to keep timing accuracy high even when the system is not in the active mode, another solution is adopted. In low-power mode, the high-frequency timer interrupts are not generated. However, the hardware timer is still ticking. First, time spent in sleep mode (in milliseconds, with rounding error) is calculated each time the MCU wakes up after sleep mode. Second, in every time the timer generates overflow interrupt (at exactly 16 seconds in sleep mode) a correction is applied, comparing the calculated value with the exact value (i.e. 16 000 milliseconds). This is implemented in a few small blocks of code and is additionally using only 4 bytes of RAM storage for file-scope variables.

Losing short-time accuracy is the tradeoff of MansOS solution. 3 millisecond error is not noticeable by a human directly (for example, when used to calculate time when to sample environmental monitoring sensors), but might make a functional difference when, for example, time slots for high-contention TDMA MAC protocol need to be allocated with high precision (arguably, not a typical application-level task). In such a case, the application programmer is better off by using hardware timers directly. In contrast, there are no long-term penalties. When 32-bit millisecond counter is used, the algorithm correctly runs up to 49 days; when 64-bit counter (enabled by a configuration option): up to millions of years.

Another tradeoff is the fact that an additional hardware timer counter must be reserved for the correction timer. However, as described above, Tmote Sky has 10 independent hardware counters, and two of them are used by the operating system in any case (one for counting ticks, one for low-power mode). Enabling the correction timer merely means that three rather than two out of 10 timers are going to be used.

2.5.3 Memory allocation

Initially the memory for system-level objects (sockets, files, queued packets) in MansOS was allocated statically, using file-scope array variables. The size of the arrays was either guessed by the system developer, or was determined in configuration files of applications. It was hard to extrapolate the correct size that a "typical" application is going to need, therefore the memory was often either wasted or allocated insufficiently.

In order to solve the problem, a simple but ingenious solution was discovered: make the object's user responsible for memory allocation for that object. Now all respective object initialization functions take a pointer to the object as an argument. For example, instead of POSIX `FILE *fopen(const char *path, const char *mode)` function now there is `FILE *fopenEx(const char *path, const char *mode, FILE *result)` function with three arguments instead of two; instead of POSIX `int socket(int domain, int type, int protocol)` now there is `int8_t socketOpen(Socket_t *socket, SocketRecvFunction callback)` that takes a pointer to the socket structure as an argument. Similarly, allocation of queued packets is now done at MAC protocol level. Each protocol driver can reasonably well estimate how large packet queue it is going to need. For most of MAC protocols, a buffer of just one packet is sufficient.

All threads are also allocated statically, avoiding the potential reliability problems inherent in systems like Mantis, which allow dynamic memory allocation for new threads at run-time, and making the code more amenable to static analysis and checking.

However, dynamic memory allocation as such is also available in MansOS. While not used in the system code in order to avoid compromising reliability, it can be used as part of application code, if the user needs it. For example, the user may want to port existing code that uses dynamic memory allocation to sensor device hardware. Availability of dynamic allocation API makes her task easier.

2.5.4 TinyOS-like design elements and MansOS: a discussion

MansOS includes some TinyOS-like design elements (for example: (1) component selection as part of application building, (2) static memory allocation, and (3) pruning of unused code). The assumption behind this is that these elements should be implemented in plain C, so that they become available to users who do not known nesC programming language. In this way, the usability of existing WSN OS design elements can be increased.

In this section we briefly discuss some elements that are partially implemented, point out what is still missing, and discuss implementation ideas for future work.

2.5.4.1 Parametrized hardware components

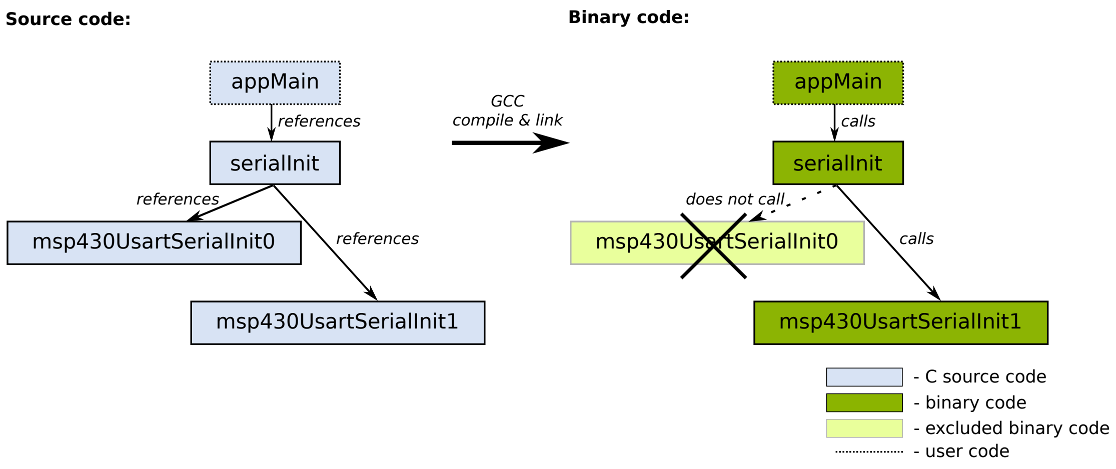


Figure 2.15: Parametrized hardware interface optimization

Some of TinyOS design elements are partially implemented. For example, an abstraction similar to parametrized hardware components are used to ensure smaller binary code sizes. While in TinyOS the functions for separate hardware components are generated by the nesC compiler, in MansOS they are written by hand. To take a concrete example, consider an application that prints characters using USART #1 serial interface on a TelosB node. (There are two, USART #0 and USART #1 hardware interfaces on this node; both are supported by MansOS software drivers.) Such an application requires that the USART module is initialized in serial protocol mode. In TinyOS, the task is done by `HplMsp430Usart1P__Usart__setModeUart()` function that is generated by nesC compiler. Since USART #0 is not used, the resulting application code is smaller than if a generic configuration function was used. MansOS achieves the same result by providing two different functions: `msp430UsartSerialInit0()` and `msp430UsartSerialInit1()`, and a high-level, inline wrapper function `serialInit()`. If GCC linker optimizations are enabled, the resulting code is as small as in TinyOS, because `msp430UsartSerialInit0()` code is optimized away (i.e. excluded from the binary image) (Fig. 2.15). Essentially, the idea is to design the code in such a way that it can be optimized by the common GCC, rather that to write a custom optimizing compiler.

2.5.4.2 Differentiating execution contexts

Similarly to desktop operating systems, there are different contexts in which code may be execution: for example, the interrupt context (inside interrupt handlers), which is characterized by limited stack space available and (usually) disabled interrupts; the system context, which usually can be interrupted by hardware events, but cannot be preempted by a user application; and the user application context, which is the least privileged of all. TinyOS explicitly separates, at syntax level, between *event handler* code (the interrupt context) and *task* code (interruptible by events, but not interruptible by other tasks). In contrast, at the moment different execution contexts (such as the interrupt context, the system code context, and the user code context) are not recognized in MansOS. If such a recognition was implemented, it would be possible to automatically check that appropriate synchronization primitives are used; for example, atomic sections rather than mutexes must be used in interrupt context. It would also allow to enforce the requirement that whenever variables variables that can also be accessed from interrupt context are accessed from system or user code context, they are accessed with interrupts disabled; this would lead to higher reliability [Levis 2004b].

Differentiation of execution context at syntax level would also be in accordance with general usability guidelines that “different things should look different” [Nielsen 1994]. Code annotations may be one way of implementing this.

2.5.4.3 Memory allocation for global objects

The memory requirements of embedded applications can frequently be known in advance. Therefore static memory allocation is possible. Levis [Levis 2012] praises the static memory allocation implemented in TinyOS as one of its most worthy features. He mentions two core benefits:

1. Efficiency. It allows to waste no additional memory for meta information.
2. Reliability. Pointers may be avoided for memory allocation, therefore the “dangling pointer” problem is frequently avoided as well.

To see how this works in detail, consider the problem of how to allocate global, system-wide objects. (A short preliminary discussion: the kernel of the operating system should know the addresses of all open sockets, all active timers, all open files etc. Furthermore, it makes *some* sense to assume (as does TinyOS) that the ratio of active objects to all objects is close to one (e.g. almost all sockets are open all the time), therefore the system code can be simplified by processing not only active objects, but all objects in a unified way.)

There are two straightforward ways how to make these global objects accessible by system code: by putting them in a *global list*, and by putting them in a *global array*.

It is easy to see that in the "global list" approach a single misbehaving object may wreak havoc on the whole system, simply by setting an invalid value as its `next` pointer (the pointer that should reference either the next object in the list, or be set to *NULL*). In TinyOS, all system-wide objects are placed in a continuous memory array, therefore this problem is avoided. (This is the "reliability" point brought up by Levis.)

It is equally easy to see that the `next` pointer is going to take up memory: at least two bytes per object. (This is the "efficiency" point.) In contrast, the array approach allows to pack the objects as densely as possible – but only if their count is known at compile time! (Once again, an assumption is made, following TinyOS, that this count *may* be known in advance, i.e. at compile time.) TinyOS nesC compiler provides a predefined function for this purpose: `uniqueCount()` that allows to know the exact number of a specific type of objects used in the applications at compile time.

How does MansOS fare at the moment? Similarly to the other C-based WSN operating systems analyzed, MansOS puts objects (such system timers and sockets) in global lists. (See Section 2.5.3 for more details on this memory allocation for global objects.) This means that MansOS faces the two problems described by Levis.

Firstly, there are no compile-time checks that the user is doing something obviously incorrect, e.g. inserting a function-local variable in the system timers list and returning from the function. To mitigate this issue, MansOS implements and uses `bool isStackAddress(void *p)` function that check whether `p` is allocated in the stack. It causes an immediate abort in the case described.

Secondly, memory usage is slightly larger compared to the array approach since a single pointer is required as a part of the structure. However, this is a tradeoff for increased run-time efficiency. TinyOS work less efficiently in some cases, for example, when there are many timers and at least one of them fires frequently. This operating system processes (and changes) all the timers used, even those that are inactive. MansOS, in contrast, processes only the initial part of the timer list. Of course, it in turn works less efficiently if there a lot of timers and proportionally the number of timers are rescheduled, and are not the first in the list. However, for the second problem there is an easy workaround at the application level: instead of rescheduling the timer, allow it to fire, and then use a counter to count all the times it would have been rescheduled. For the first problem there is no such workaround, as both (a) a lot of timers and (b) at least one frequently running timer are simultaneously required by many WSN applications.

The “global array” (rather than “global list”) approach was not implemented in MansOS, because, once again, it requires to know the exact count of statically allocated run-time objects (like timers or sockets) at compile-time. TinyOS relies on the nesC compiler to do that. However, it is possible even without using nesC. One option is to use template metaprogramming present in C++. By using it, a counter that computes that number of instantiations of a specific object can be implemented. Such a counter can do instantiation count across even multiple compilations units, because C++ template macroprogramming is a Turing-complete compile-time computation method. Unfortunately, C preprocessor is less powerful. It provides `__COUNTER__` macro, but the macro allows to count instantiations only in a single compilation unit.

Actually, the “global array” approach is already implemented in MansOS, but for just one type of system objects: threads. We believe this approach is more appropriate for threads compared with other system objects. Why? The number of threads is expected to be small (Section 2.4.1) even compared with the number of timers and sockets, therefore MansOS assumes that it is efficient to loop through all threads (Algorithm 2.3) every time the system scheduler code is executed. There is also the requirement that the maximal number of threads must be set in configuration file, therefore it is always known at compile time.

2.6 Evaluation

In this section, aspects of MansOS are experimentally evaluated and compared with aspects of TinyOS, Contiki, and Mantis. LiteOS and Arduino are not included in this comparison because they lack TelosB support. The versions used for evaluation were: MansOS revision 918 (May 2013), TinyOS revision 6033 (December 2012), Contiki git commit *f5c8 . . .* (October 2012), and the latest public release of Mantis, 1.0-beta (October 2007).

2.6.1 Source code compactness

Four applications were implemented in each operating system: *Active* – busy looping application, *RadioRx* – radio packet reception, *RadioTx* – radio packet transmission, *Combined* – a sense-and-send application that also writes sensor values to the external flash memory[1].

First we compared source code size of the *combined* application in all implementations. (*Contiki-optimized* is an implementation that will be discussed in the next section. It has optimized resource usage, rather that source size.) The size was evaluated excluding comments and empty lines, but including configuration files. The source of nesC and Contiki applications were formatted by using TinyOS and Contiki example applications as basis, respectively, while C applications were formatted according to MansOS coding guidelines[2].

[1]The implementations of these applications are available at `http://mansos.edi.lv/dissert/testapps.tgz`

[2]Available at `http://mansos.edi.lv/wiki`

Source code for the *combined* application's event-based implementation in MansOS is also given in Listing 2.4.

Listing 2.4: The *combined* application

```c
#include <stdmansos.h>
// define sampling period in miliseconds
#define SAMPLING_PERIOD 5000
// declare our packet structure
struct Packet_s {
    uint16_t voltage;
    uint16_t temperature;
};
typedef struct Packet_s Packet_t;
// declare a software timer
Alarm_t timer;
// declare flash address variable
uint32_t extFlashAddress;

// Timer callback function. The main work is done here.
void onTimer(void *param) {
    Packet_t packet;
    // turn on LED
    ledOn();
    // read MCU core voltage
    packet.voltage = adcRead(ADC_INTERNAL_VOLTAGE);
    // read internal temperature
    packet.temperature = adcRead(ADC_INTERNAL_TEMPERATURE);
    // send the packet to radio
    radioSend(&packet, sizeof(packet));
    // write the packet to flash
    extFlashWake();
    extFlashWrite(extFlashAddress, &packet, sizeof(packet));
    extFlashAddress += sizeof(packet);
    extFlashSleep();
    // reschedule our alarm timer
    alarmSchedule(&timer, SAMPLING_PERIOD);
    // turn off LED
    ledOff();
}

// Application initialization
void appMain(void) {
    // wake up external flash chip
    extFlashWake();
    // prepare space for new records to be written
    extFlashBulkErase();
    // initialize and schedule our alarm timer
    alarmInit(&timer, onTimer, NULL);
    alarmSchedule(&timer, SAMPLING_PERIOD);
    // enter low-power mode
    for (;;) sleep(1);
}
```

The results (Fig. 2.16) suggest that compared to other WSN OS, MansOS allows to write applications with the same functionality using shorter code. The difference is rather small and inconclusive compared to Contiki and Mantis, but more than two times compared to TinyOS. This is an important usability benefit of the system, because shorter code is more easy to understand and manage (at least when

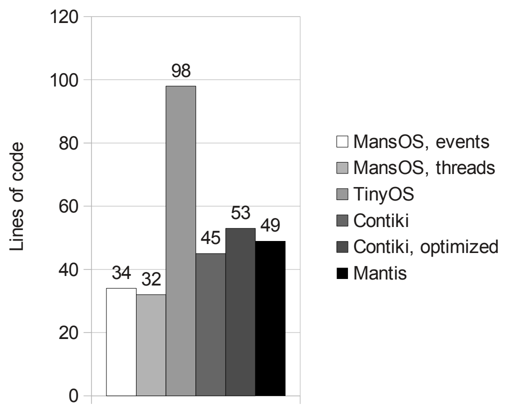


Figure 2.16: Source code size comparison of the *combined* application

all the other aspects, such as familiarity and complexity of the code are equivalent). In contrast, lengthy source code in TinyOS signals a potential usability problem. We point out that even though TinyOS applications are written in a different programming language (nesC), the level of abstraction of TinyOS code is roughly the same, since both C and nesC are high level languages. Even more, since nesC is a superset of C [Gay 2009], it allows for *more* concise syntax in some cases, rather than the opposite.

Further analysis is required to determine whether the complexity per line is small enough in TinyOS to balance out the additional code size.

2.6.2 Binary code size and RAM usage

The RAM usage and code sizes were obtained by compiling with *msp430-gcc* version 4.5.3, using the default optimization level of each operating system. For MansOS applications, 256 byte stacks were used.

There were four versions of MansOS and two versions of Contiki applications. First we discuss the difference and motivations between “release” and “default” MansOS versions. Since WSN applications are often prototyped on more powerful devices than are used for the final deployment, we also consider a size-optimized (“release”) versions of MansOS applications. To build such a release version, only one change was required: addition of `USE_PRINT=n` configuration option. As the motes in the deployed WSN usually cannot be monitored using serial interface, such a change is natural at the end of software prototyping, when it is prepared for the release. (If wireless debugging is required in a deployed WSN, MansOS also offers a configuration option to redirect debug output to radio.) Certain parts of debugging code (e.g. the *ASSERT* macro) remained even in the “release” versions.

Now we turn to the question of Contiki. The applications were initially implemented using the default, limited component selection mechanism present in this operating system. `nullmac_driver`, `nullrdc_driver`, and `framer_nullmac` options were selected for all of the applications, and `CONTIKI_NO_NET` define was also selected in `Makefile` of the *loop* application. Then, in order to show the im-

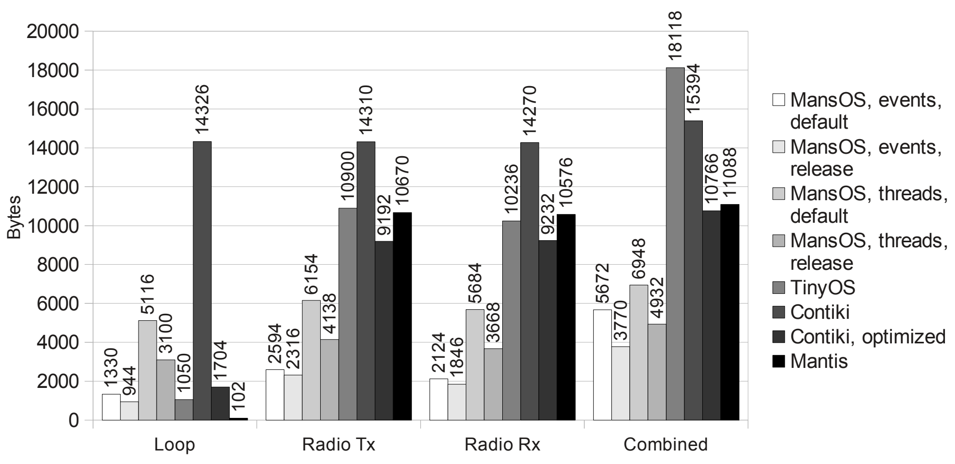


Figure 2.17: Application binary size comparison

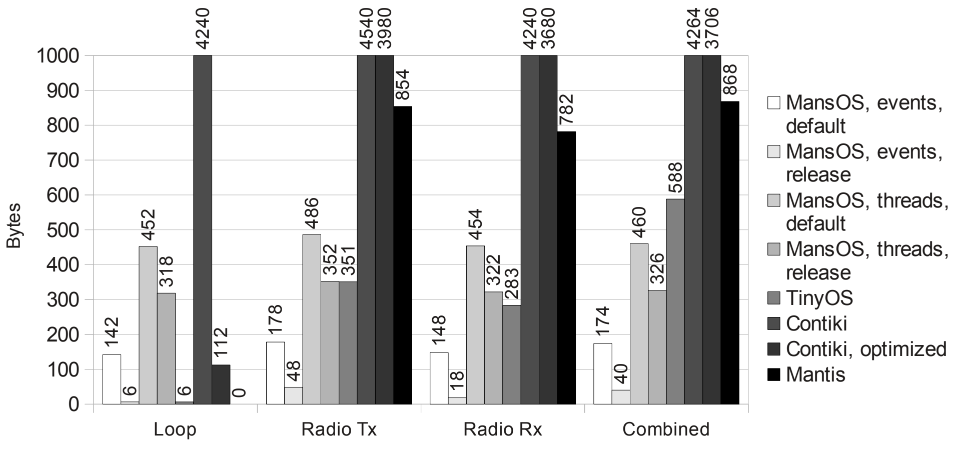


Figure 2.18: Application RAM usage comparison. Contiki RAM usage is not shown in full

provements possible in this default configuration mechanism we also ported the applications to the optimized version of Contiki that partially incorporated MansOS component selection mechanism (Section 2.3), leading to "optimized" versions of the same applications.

The results are given in Fig. 2.17 and Fig. 2.18. As for code sizes (Fig. 2.17), MansOS shows the best results in three of four test cases, the only exception being *loop* program: in Mantis it uses only 102 bytes, compared to 944 bytes in MansOS release version, without threads. In Fig. 2.18 Contiki RAM usage is not shown in full; for all applications it is between 4240 and 4540 bytes for the unoptimized version and (excluding *loop*) between 3680 and 3980 bytes for the optimized version.

All of the WSN OS analyzed try to reduce binary code size in some way. MansOS: by using the configuration mechanism (Algorithm 2.1 and Algorithm 2.2) and by enabling linker optimizations, essentially achieving garbage collection of unused functions; Mantis: by building separate components as libraries and linking them together (allows to discard unused libraries), TinyOS: by topologically sort-

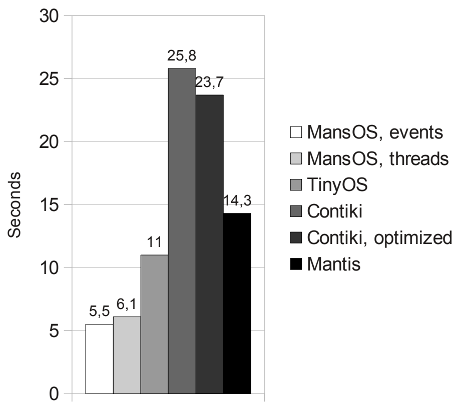


Figure 2.19: Application compile & upload time comparison of the *combined* application

ing all functions in source files and pruning unused ones from the final binary code. Contiki enables function-granularity garbage collection (implemented by the GCC compiler and linker themselves), but demonstrates the worst results of all OS, as it also enables most of components by default.

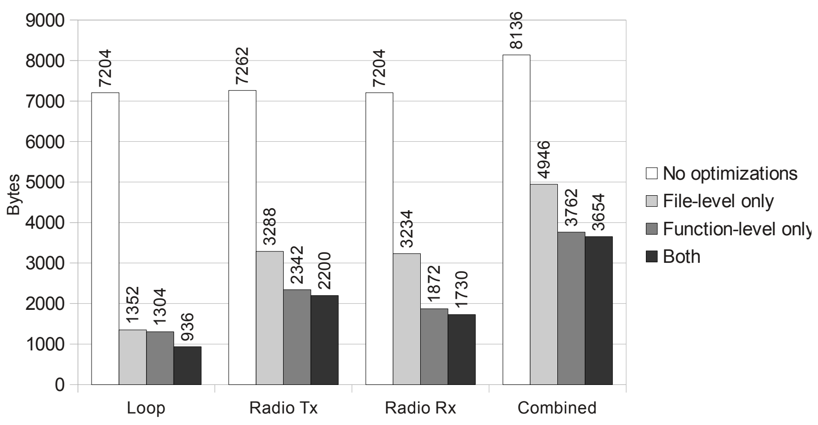
MansOS compared with Contiki and Mantis additionally benefits from the component-granularity link time optimization (Section 2.3), and by parametrized hardware component optimization (Section 2.5.4). The effects caused by disabling these techniques are show in Fig. 2.20 (for MansOS; this, in contrast to the rest of results in this chapter, was evaluated on MansOS revision 981) and Fig. 2.21 (for Contiki). In MansOS, enabling function-granularity optimization also caused unused parametrized hardware components to be optimized away. In contiki, function-granularity optimizations were already present in the mainstream version, so only componenent-granularity optimizations is considered. The effect of this optimization is comparatively much larger in Contiki than in MansOS, because only the optimized version of Contiki does not enable and use all of the components by default.

The effects of file-granularity optimization are more pronounced on simpler applications, as the *combined* application uses most of the components enabled by default, while the simpler ones do not. Therefore, they may be optimized away.

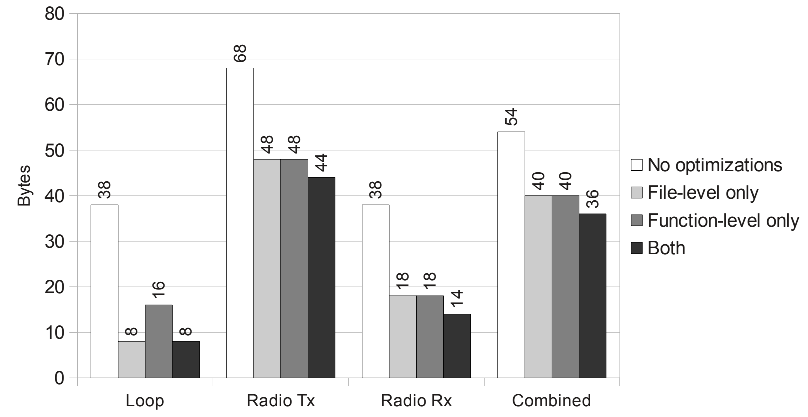
Interestingly, judging by the publication [Cao 2008] LiteOS also achieves small application code size. LiteOS applications use system calls, which allows to include only the user logic in the code of the application itself, rather than to build a monolitic binary image that includes the whole kernel functionality.

Larger binary code size in TinyOS is partially caused by limitations in this OS hardware abstraction model: direct access to radio chip's driver code is prohibited and Active Message interface has to be used.

The optimized version of Contiki demonstrates good results for this set of applications (Fig. 2.21): at least 30.1 % reduction in flash usage and at least 12.3 % reduction in RAM usage, showing that the MansOS component selection mecha-



(a) Flash



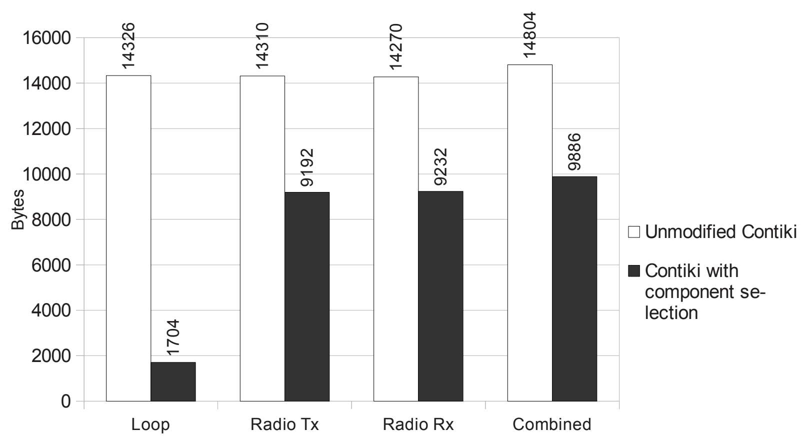
(b) RAM

Figure 2.20: Effects of enabling link-level optimizations in MansOS (event-based, “release” version)

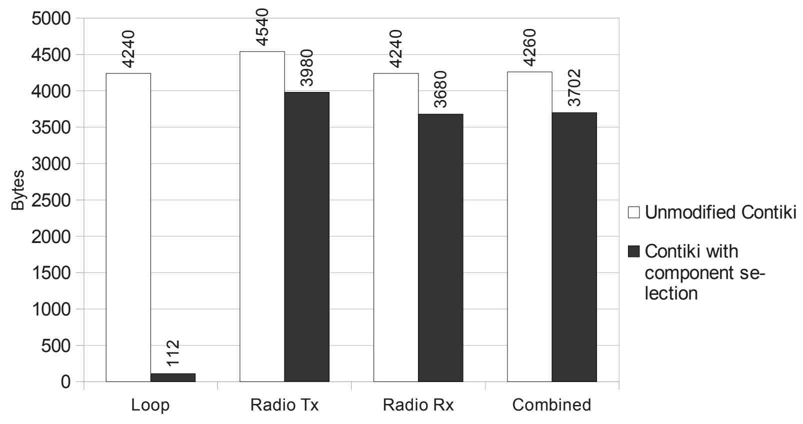
nism is a viable way how to reduce superfluous resource usage in other operating systems as well.

As for Mantis, their approach is efficient, but suffers from usability problems. A number of changes are required to build their latest release with the current GNU compiler version, including defining `putchar()` as dummy function in user code and commenting out multiple references to `mos_led_display()` function in kernel code. The problems are caused by circular dependencies of the libraries. We can conclude that increasing the number of separately linked components is detrimental to the usability of the core system, since the number of inter-component dependencies grows too fast.

Shorter binary code size leads to tangible benefits for the WSN OS user. Firstly, energy requirements in reprogramming are directly proportional to the code size, if full run-time reprogramming is used. Every byte of code transmitted through the air requires spending a small, but substantial amount of energy. Even though all of the analyzed OS allow some kind of partial run-time reprogramming, it is not always sufficient. Whenever core parts of the system are changed, full reprogramming is



(a) Flash



(b) RAM

Figure 2.21: Effects of enabling link-level optimizations in Contiki

still required. Therefore, the number of bytes that need to be transmitted ought to be kept small.

Secondly, smaller code leads to shorter development times, as putting the program on sensor devices becomes faster. To show this, we measured combined compilation and upload time on TelosB platform (Fig. 2.19). The arithmetical average of three measurements was used; the standard deviation was too small to show it in the figure, no larger than 1.1 % of the corresponding average value. Faster upload is important because *code-and-fix* approach is currently typically used in the sensor network software development process [Picco 2010]. Due to limited WSN debugging options, when debugging a specific problem, sometimes it is required to reprogram the device large number of times to test a specific hypothesis about the problem. The author has spent hours doing this kind of debugging. If the reprogramming process can be made just 20 seconds faster, the length of a debugging session sometimes can be reduced by half an hour. (The measured average difference between Contiki and MansOS without threads was 20.33 seconds.) Also, if the number of sensor nodes for a deployment is great, their programming time can be significantly

Table 2.3: Flash memory usage in the *combined* application, bytes

Subsystem	**No threads**	**With threads**
Kernel	952	1090
Radio	1064	1288
USART & SPI	412	468
Flash	544	544
ADC	154	154
LEDs	38	94
Arch & platform	310	310
Compiler & library code	142	244
User code	154	152
Threads	0	588
Total	**3770**	**4932**

Table 2.4: RAM usage in the *combined* application, bytes

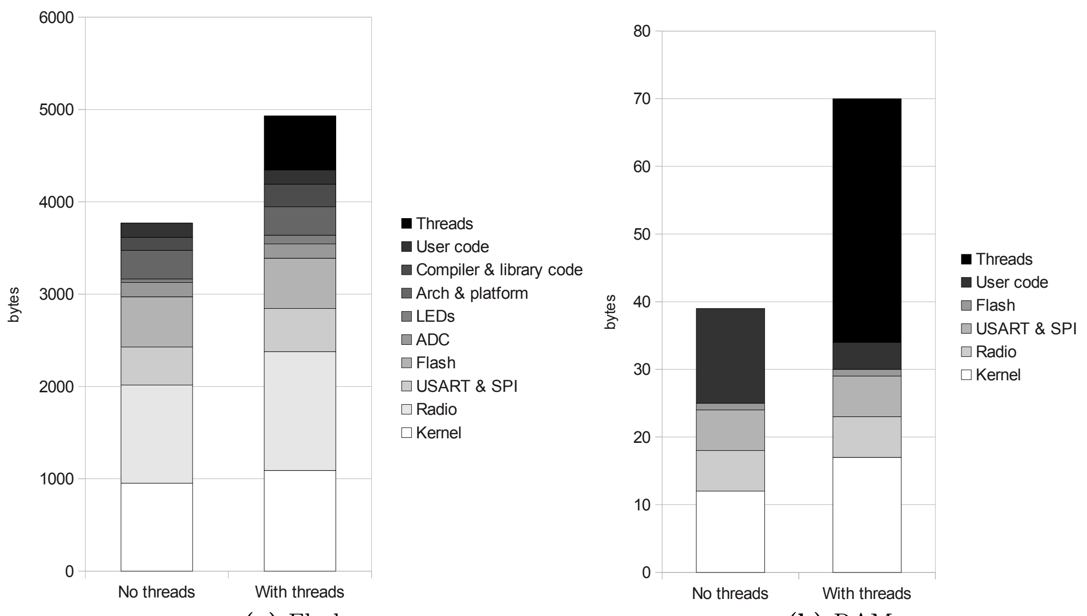
Subsystem	**No threads**	**With threads**
Kernel	12	17
Radio	6	6
USART & SPI	6	6
Flash	1	1
User code	12	17
Threads	0	36
Total	**39**	**70**

reduced. If 20 second shorter binary code upload takes place, for a network of ten nodes, programmed sequentially, the difference is 3 min 20 sec in total, but for 100 nodes: more than 33 minutes. (The sensor devices typically cannot be programmed in parallel, as this process requires programming hardware; the number of these hardware programming devices is not likely to scale close to the number of sensor devices.)

Furthermore, building MansOS programs is faster than their counterparts in other OS, because MansOS configuration mechanism excludes most of unnecessary source files from the build by default. TinyOS approach is efficient in this regard as well – we hypothesize it's because all *nesC* files are pre-compiled to a single C file for fast processing.

The MansOS in event-based form takes considerably less flash space than the threaded version. The difference is mostly due to the complexity of the thread implementation itself (Table 2.3). While using more resources in general, the threaded version leads to shorter user code and smaller RAM usage in it, because smaller state information has to be kept inside application's logic.

RAM usage is given without including memory allocated for stacks (512 bytes for each thread by default, 256 bytes configured for this test). Even though comparatively large amount of memory is used in this way, it seldom would cause problems for real applications, because code memory, not RAM, is the scarcest resource on Tmote Sky. This is evidenced by the example application (Table 2.3, Table 2.4, Fig. 2.22), because it uses proportionally more of total code memory (4932 bytes of 48 kB, i.e. 10.0 % of it) than of total RAM (70 + 256 bytes of 10 kB, i.e. only 3.2 % of it).



(a) Flash **(b)** RAM

Figure 2.22: Subsystem resource usage in the *combined* application

2.6.3 Thread implementation

Thread implementation is compared with Mantis, because of the four operating systems considered only MansOS and Mantis enable preemptive multithreading by default. Both include support for theoretically unlimited number of threads, although in MansOS the upper bound must be specified at compile time, and usually is small. The thread structure takes 12 to 16 bytes in RAM in MansOS (minimal and maximal configuration) and 21 bytes in Mantis. The maximal number of total threads in Mantis is 10, defined as a system-level constant. Since one of those threads is a system thread, the maximum number of user threads in Mantis is 9. In MansOS the number of user threads is defined as a application-level constant, and by default is just 1.

Two distinctive features of MansOS thread implementation are apparent:

- Lower resource requirements (Fig. 2.23). The scheduler used in MansOS is simpler, e.g. it has no separate queues for ready and sleeping threads. This tradeoff is justified by the low number of threads typical in a WSN application (see Section 1.4.4.4), which allows the scheduler to process all threads in each context switch. Furthermore, in WSN OS user threads can be expected to be

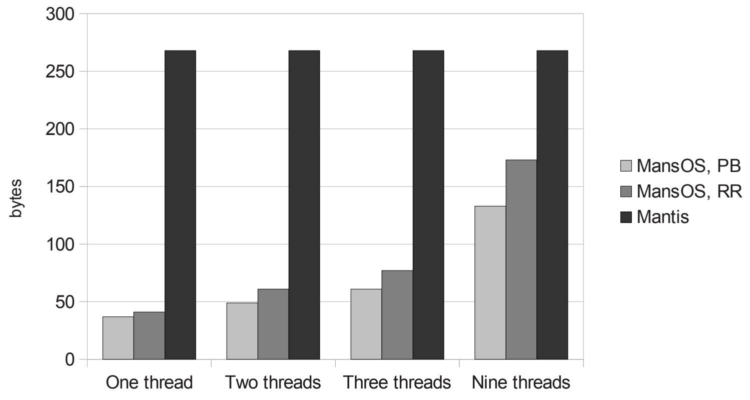


Figure 2.23: RAM usage comparison of the thread module relative to the number of user threads. *PB*: priority based scheduling, *RR*: round-robin scheduling

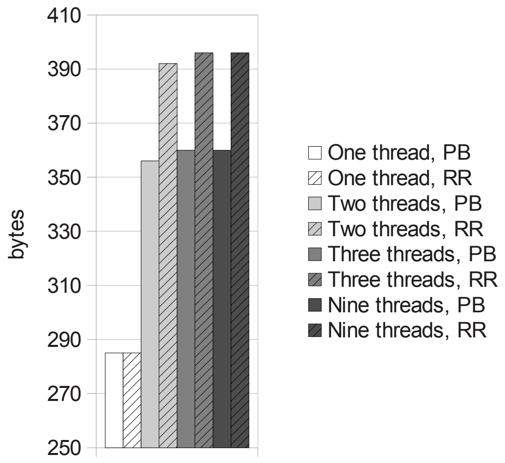


Figure 2.24: Flash memory used by `schedule()` function relative to the number of user threads. *PB*: priority based scheduling, *RR*: round-robin scheduling

cooperative (because for a single-application WSN, all user threads belong to the same application), so the fairness of the scheduler becomes an optional, rather than a critical requirement.

- Better adaptation (Fig. 2.23, and especially Fig. 2.24). Mantis memory requirements are constant and do not depend on the number of threads used (the flash usage changes in the figures are due to longer user code only). MansOS requirements are flexible.

The RAM usage was evaluated without including stacks. The required stack size is dependent on compiler version and on system libraries being used. MansOS leaves large safety margin and uses 512 byte stacks by default (except on low-memory platforms); the choice is motivated by large stack space requirements of library functions. For example, the `PRINTF` macro in MansOS eventually calls *libc* functions. The macro, when called without arguments, already uses 62 bytes of stack space (*msp430-gcc* version 4.5; the usage is even higher in *msp430-gcc* version

4.6). The arguments passed to `PRINTF` can easily use ten or more bytes additionally. Therefore, future work includes implementing formatted print in the OS itself and in a more optimized fashion, as is done in Mantis. On the other hand, Mantis allocates stacks in heap, so more than the amount pictured in Fig. 2.23 is used.

The heart of the thread implementation in MansOS is the `schedule()` function that selects which thread to run next. The binary code size of this function depends on the number of threads used (Fig. 2.24). Round-robin based scheduling is more costly, partially because 32-bit *last-time-run* values are used instead of 8-bit priority values, and partially because thread's *last-time-run* is updated every time a thread is run, while priorities are kept unchanged during whole execution time. However, in all cases the resource requirements can be easily satisfied by a typical WSN sensor node.

2.6.4 Execution flow analysis

Taking into account the lower binary code size of MansOS applications compared to TinyOS, analysis is needed to convince the reader that MansOS is not missing some essential functionality that would render its application useless in real-world conditions.

When tracing the execution, one discovers that both operating systems do similar tasks:

- initialize the watchdog;
- calibrate the digitally-configurable oscillator;
- initialize hardware timers;
- initialize LEDs;
- put the external flash chip in deep sleep mode;
- initialize the list (MansOS) or array (TinyOS) of software timers;
- run the core function / scheduler / loop.

In MansOS event-based version, `appMain()` is used for user initialization only and for entering an infinite loop which calls `sleep()`. The real work is done by a timer callback function that is called repeatedly by the system alarm list processing code.

In MansOS multithreaded version, an array of two threads is initialized, and the user thread is started. (System thread is executed in the main execution context.) The user thread's start function is `appMain()`, which never returns, but performs all the work and calls `sleep()` in a loop.

In, TinyOS a task loop runs all tasks that are scheduled and are ready to run. If no tasks are scheduled, the scheduler puts system in a low-power mode. Each interrupt in turn causes the associated event handler with it to execute; these handlers may schedule tasks for later execution.

The MansOS code usually is smaller not because it is missing some critical functionality, but because TinyOS offers much more options to the user (in ADC control and radio control most prominently for this example). Other that that, TinyOS also includes this extra functionality:

- code for 8 hardware timers by default, even though not all of them are used by the application;
- code for CSMA access of the CC2420 radio, while MansOS uses it at the PHY layer directly;
- code for Active Message creation and management, which MansOS send out data in an untyped C structure;
- code for resource arbitration (has to be done partly manually in MansOS).

The extra RAM usage in TinyOS is mostly because of arbitration code: more variables are required to keep in track the state of the system. Also, some components (such as CC2420 radio driver) use run-time variables where MansOS allows only compile-time changes (for example, whether to ACK automatically, whether to do address recognition, etc.). In several cases the state is stored in parallel to hardware, which also stores the same state (for example, for radio channel). There is also a buffer for radio packet reception, even though it is never required by the application logic.

From all of this, only manual calls to `sleep()` and manual resource arbitration (avoided in chip drivers that were implemented later) decrease usability. None of the problems make MansOS impractical to use.

2.6.5 Energy consumption evaluation

Experimental setup. We measured the energy consumption of the *combined* application in one sensor read period (5 seconds) with PowerScale ACM probe, using 20 kHz sampling frequency (maximal measurement error: $\pm 0.5\,\mu$A by datasheet). We used a simple, synthetic application on purpose, in order to keep the number of experimentally controlled variables manageable and the analysis accurate. We also disabled all MAC-level activity, which turned out to be the biggest energy-consumer in a preliminary evaluation. Although radio packet transmission is present in this application, the long-term average energy consumption is MCU-bounded rather than radio-bounded.

We compared MansOS with TinyOS and Contiki. Mantis was not included in this test because even simple Mantis applications that were compiled for TelosB with a recent GCC compiler (mp430-gcc 4.6.3) failed to start (including on a Tmote Sky sensor node). However, existing research [Duffy 2008][Lajara 2010] already has demonstrated that Mantis is not as energy efficient as TinyOS.

The test applications were running on a single AdvanticSYS XM1000 sensor node with on-board light and humidity sensors. It had MSP430F2618 MCU clocked at 8 MHz active-mode frequency. Although it is known [Sieber 2013] that sensor network nodes show significant variance in energy consumption, we did not include more than one device in the test because we were primarily interested in the relative performance of the software under test rather than absolute consumption values.

In TinyOS, the listening interval of the low-power duty cycle MAC protocol provided by this system must be configured from the application; the maximum interval between consecutive listening periods is 64 seconds, and listening cannot be disabled completely, unless the low-power listening component is modified or replaced. By asking to replace an integral part of the system such as a MAC protocol in order to optimize energy usage, TinyOS puts a rather stringent requirement on the users. In any case, for this test we modified the core code of the existing implementation in order to disable listening altogether.

In Contiki, the behavior of the default MAC protocol also can be optionally be configured from the application (rather than *must* be configured – in contrast with TinyOS). We changed the MAC protocol back to the default (from `null_mac` to Contiki MAC), as well as disabled all other custom configuration options; we also added a single line of code to the C code of the Contiki application to turn MAC protocol off. The sensors also had to be turned on and off manually (see the discussion below), which required additional 4 lines of code.

There is one more, qualitative aspect of energy usage that should be discussed before review of quantitative results. An operating system is easier to use if it does the resource management more implicitly. With regard to this criteria, there is no clear winner. All of the OS make *MCU* power management implicit, except MansOS without threads, which requires explicit calls to `sleep()` function. Although this usage pattern (demonstrated in Listing 2.4) is not complicated, it still puts some effort on the user. All of the OS also make *radio* power management partially implicit: after sending is completed, the radio enters the idle mode automatically. However, TinyOS requires explicit low-power listening configuration; otherwise radio is constantly in the listening mode. TinyOS makes *external flash chip* power management implicit; in MansOS it is explicit, and Contiki follows "do not care" policy and offers no API for this purpose at all. Finally, MansOS and TinyOS make *ADC-based sensor* power management implicit, but Contiki requires explicit calls to turn the sensors off. When porting the *combined* application between platforms, we used the explicit API where necessary; in contrast, we did not use explicit hardware-level commands where no API was provided. Therefore, the external flash memory chip was not turned off on Contiki platform.

Results. The average, minimal and maximal current consumption of a single sensor reading period is given in Table 2.5 and in Fig. 2.25. The results are average of three samples; the standard deviations between these samples are also included in the table. Several interesting values are highlighted.

At the first approximation, for all of the operating systems, the energy consumption pattern for this application looks like a single peak (repeated every 5 seconds), while the absolute majority of the time is spent in a low-power mode (below 1 mA) (see also Fig. 2.26). The peak corresponds to the time interval when LED is lit, ADC sampled, data written to flash and transmitted to radio.

The length of the peak period is approximately 3.7 milliseconds for Contiki and 5.4 to 5.7 milliseconds for MansOS. It is only 1.5 milliseconds for TinyOS; however, the slope at the sides of this peak is not steep, and increased energy consumption

Table 2.5: Average, minimal and maximal current consumption of the *combined* application. In brackets: standard deviation of the three samples, expressed as % of value

	Average	Min	Max
MansOS w/o threads	21.8 μA (1.86 %)	10.2 μA (0.28 %)	25.7 mA (1.17 %)
MansOS with threads	28.2 μA (1.54 %)	10.2 μA (0.28 %)	25.8 mA (1.05 %)
TinyOS	69.9 μA (**17.41 %**)	10.8 μA (0.27 %)	25.5 mA (0.30 %)
Contiki	**787.2 μA** (0.02 %)	**231.4 μA** (1.98 %)	26.1 mA (1.66 %)

(more than +3 μA compared to the baseline) is present for up to 10 milliseconds.

Almost all of the rest of the time was spent in low-power mode (with MCU and radio turned off). The typical current consumption in this sleep mode was between 10 μA and 20 μA for MansOS and for TinyOS (slightly smaller for MansOS), but 750 to 770 μA for Contiki.

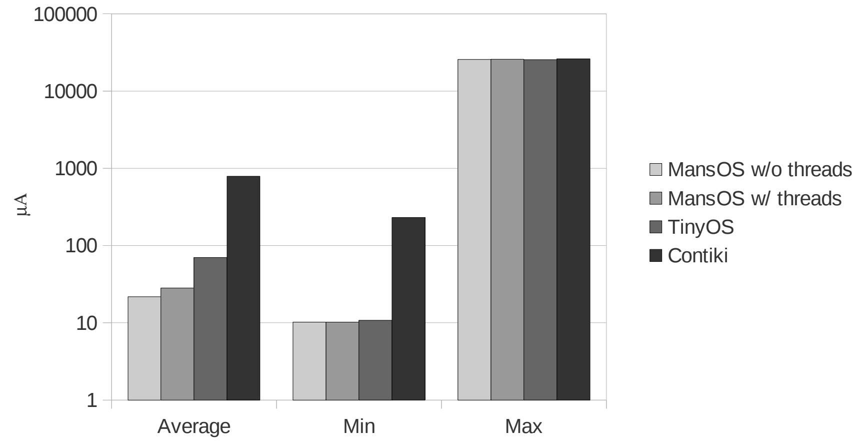
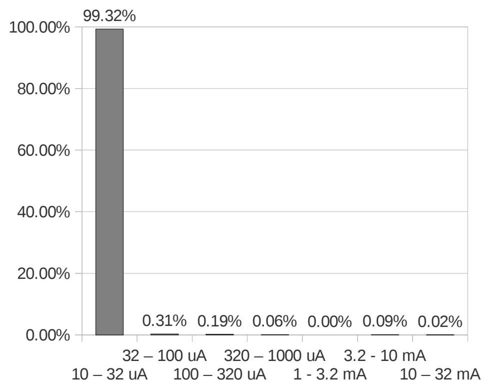


Figure 2.25: Average, minimal and maximal current consumption of the *combined* application (logarithmic scale)

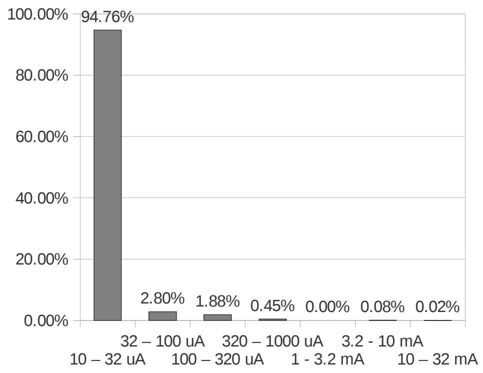
Fig. 2.26 shows the proportions of each energy usage levels in different operating systems. It can be seen that both TinyOS and MansOS spend almost all of the time in very low current consumption mode: below 100 μA. Contiki spends majority of the time consuming between 0.32 mA and 1 mA. No OS spends large proportion of time at levels above 10 mA.

Discussion. First of all, in this example application MansOS and TinyOS demonstrate average energy usage on this platform that is an order of magnitude better than that of Contiki. The average usage of MansOS is also significantly better than that of TinyOS; the average usage of MansOS without thread is significantly better that of MansOS without threads.

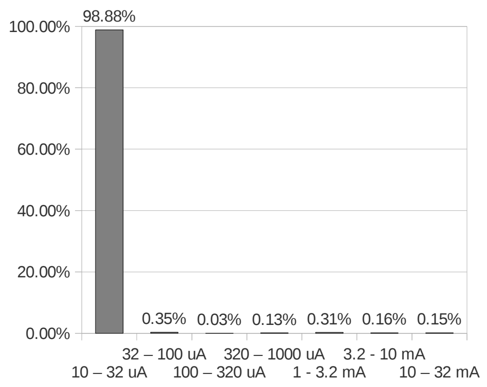
The minimal energy usage of Contiki is relatively very high, but there are no significant differences (at this level of precision) between the minimal energy usage of TinyOS and MansOS. The maximal energy consumption is similar on all OS.



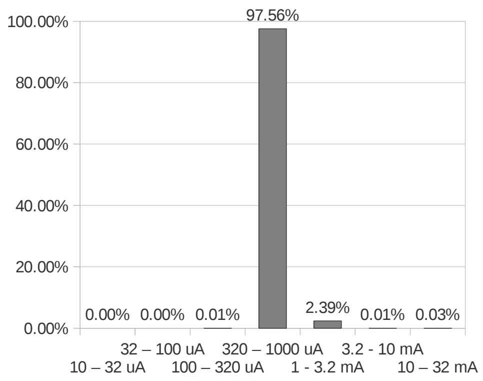
(a) MansOS without threads



(b) MansOS with threads



(c) TinyOS



(d) Contiki

Figure 2.26: Proportions of time spent in specific current consumption levels in the *combined* application

The variance of minimal and maximal energy consumption between samples is low; on the other hand, the variance for average consumption is high at least for TinyOS. More research is needed to determine the cause of this variance, as well as the reasons behind the difference between the average consumption on MansOS and on TinyOS.

This example application also demonstrates the cost of threads in MansOS. Adding threads increases average energy consumption by 29 %, although this increase in energy is going to be proportionally smaller in more computationally-intensive or communication-intensive applications. The threaded version of MansOS consumes more, because the scheduler itself is run periodically even if there are no active tasks. The times when scheduler is run are observable as small bumps in the energy consumption profile of the device; the average period between then is 100 milliseconds for this application.

It was already known [Lajara 2010] that Contiki consumes more energy than other popular WSN operating systems. This is primary because the system by default configures and leaves on everything it supports. Even though there is explicit code in this application that turns off MAC protocol and ADC hardware, high energy efficiency is still not achieved. For example, Contiki does not turn off the voltage reference generators: one is of the radio chip, one of the internal ADC ($200\,\mu$A consumption). It initializes and does not put in deep-sleep mode the external flash chip. All this leads to power consumption that is from 750 to $770\,\mu$A even in the sleep mode. Periodic low peaks are also present (up to $1200\,\mu$A, with 7 to 8 millisecond period). The lowest energy consumption values were recorded directly after radio transmission is completed.

What do these results mean? If the sensor node was programmed using this application written in MansOS and powered from a pair of typical alkaline AA batteries with 2700 mAh capacity, it would take more than 14 years to deplete the batteries. If MansOS with threads was used, the system would last 10 years and 11 months. If TinyOS: 4 years and 5 months, if Contiki: just 143 days (between four and five months). (We are aware of the limits of such an extrapolation and include it pedagogical and explanatory reasons, rather than as a scientific claim expected to hold in real-world conditions. This extrapolation is limited by many factors, e.g. it uses a simple linear discharge model, it ignores self-discharge current, which in real-world would be comparable if not bigger, and it uses absolute consumption values from results that are not validated on more than one device.)

MansOS superiority in energy efficiency is going to become smaller both as the applications get more complicated (e.g. more communication is required), and if the user adds application-specific extensions to the OS. Still, for simple applications like this the MansOS approach leads to notable gains compared to Contiki. The user who wants to write a typical sense-and-send application will be better off by choosing MansOS, as it offers higher system lifetime directly out-of-the-box.

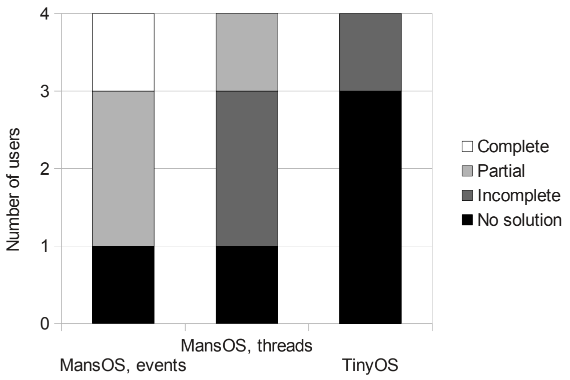
Also, even if the assumption that energy consumption in most of sensor network applications is radio-bounded rather than MCU-bounded holds, our selection of this specific application for testing is still justified. The energy consumption of radio-bounded applications is dependent primarily on network protocols. If a hypothetical application is energy-efficient in Contiki, it is so because it uses an energy-efficient network protocol stack. It is likely that these protocols can be ported to MansOS without changing the core of MansOS. (For an example see the note at the end of *Concluding remarks* of this chapter.) Therefore, it makes sense to compare energy efficiency of just this core part, rather than of the whole networking stack.

2.6.6 User study

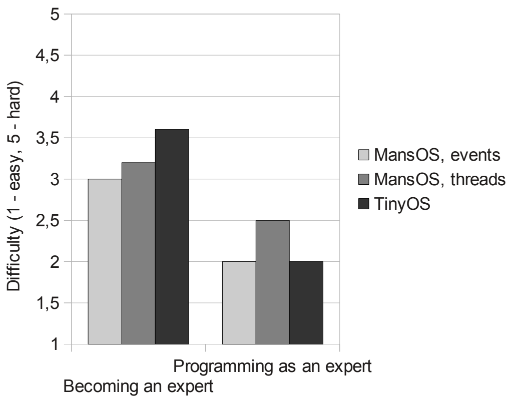
A small-scale user study was performed to compare success in sensor network application development on TinyOS and on MansOS for first-time users. The participants were third year computer science students taking a WSN course. The test took place in the first month of the course, and none of the users had previous WSN programming experience. However, all of them had been taught a C++ course before.

The task was to modify a simple sense-and-send application. The code of the application was provided to the users as a template. Originally it read light sensor with 3 second interval and sent out the values to a base station. The users had to add humidity sensor reading, change the interval to 2 seconds, and send a two-field packet to the base station rather than a single value.

The users were separated in three groups, four users in each: TinyOS users, MansOS using event-based programming model, and MansOS using thread-based programming model. According to usability testing research [Barnum 2010], four user large sample size can uncover approximately 75% of all possible findings for a particular test.



(a) Student solutions



(b) Difficulty estimation

Figure 2.27: Results of the experimental evaluation and opinion poll

The results are presented in Fig. 2.27. Student solutions were judged by their completeness and correctness (Fig. 2.27a), and the students were asked to estimate the "expert-level usability" of their respective environment (Fig. 2.27b).

Discussion. Even though the sample size is small, it is still enough to draw statistically significant conclusion that TinyOS usability for the first-time user is lower ($P < 0.02$, i.e. the null hypothesis that OS choice has no effect on usability has probability lower than 2 %; for the null hypothesis we assumed that a randomly picked student had the same, 25 % probability to achieve any of the four outcomes in the test).

The conclusion is hardly surprising, as all of the users had programmed in C++ (which is nearly backwards compatible to C) before, and none of them had programmed in nesC before. Still, the evaluation is useful, because it experimentally validates the assumption that for simple applications, for IT professionals C-based sensor network operating systems are easier to start using than TinyOS.

There were complaints about usability problems in all three groups ("limited or hard-to-understand documentation" was especially prominent); however, only in the TinyOS group they caused majority of participants end the study without creating any solution at all.

Threads and events, in contrast, have statistically indistinguishable usability in the context of the test ($P > 0.05$), although thread-based group performed worse. It is possible that this was because of the event-based nature of the task. The implementation of “thread” group used polling, which did not reflect this nature as well as the implementation of the “event” group that used callback functions.

The students who used MansOS were sometimes confused by the need to modify configuration file in addition to C source code. Apparently, the confusion was due to unfamiliarity, rather than inherent complexity of using it.

The students did not expect that using TinyOS at the “expert-level” (i.e. as confident and experienced users) is going to be significantly harder than MansOS (Fig. 2.27b), but they admitted that “becoming an expert” may take longer.

2.7 Applications

The most successful MansOS application up to date, environmental monitoring in precision agriculture, is described in details in Chapter 4. This section lists a few other MansOS applications or potential applications.

2.7.1 Wild animal monitoring

One prospective application of MansOS is wild animal (lynx) monitoring [Zviedris 2010] (see also Listing 3.29). While environmental monitoring is not a new application area for MansOS, the system requirements of mobile object monitoring in large areas with sparse connectivity are very different compared to the requirements for microclimate monitoring in an orchard. For example, wild animals are neither stationary nor cooperative, therefore human access to the data and the device itself during the lifetime of the system is problematic.

The key part of the LynxNet hardware is the collar device. It is an integrated hardware solution that consists of Tmote Mini sensor device, external radio transceiver, attached sensors (accelerometer, temperature and humidity) and a GPS device. We decided to use low-cost radio chip: LINX TRM, 433 MHz frequency, on-off keying modulation. The chip offered no software abstractions for packet-level access or byte-level access to radio data; it had to be sent and received bit by bit. Therefore the PHY level communication protocol (data coding and decoding) had to be implemented in software. Nevertheless, the radio communication worked for 300 meter distances (both line-of-sight and in a forest, where the signal propagation path was occluded by leaves). Later, forward error correction (using Hamming codes) was added to the communication protocol and MansOS; it should increase the communication distance by exploiting redundancy.

This radio communication distance is not large enough to cover a significant part of activity area of a typical lynx, but could be useful for a number of base-stations placed in the forest. The base stations should be situated in places which the lynx visits frequency. The base stations should also be equipped with mobile Internet connection in order to upload the data in a database server once they are received from a lynx.

In order to increase robustness of the system, a number of MansOS features would be used. In order reboot the system in case software freezes there is a watchdog timer. Then there is code for periodic digitally-configurable oscillator reconfiguration. The high-frequency oscillator has period that is sensitive to temperature changes, therefore in the forest conditions it should be periodically re-calibrated. MansOS allows to do it automatically.

2.7.2 Automotive applications

Vehicular sensor networks is a promising subfield of WSN, and EDI is carrying out a number of research activities in this area. MansOS has been used in two of them: road surface monitoring and vehicle mode detection. The embedded hardware platform used for both of these projects is the CarMote [Mednis 2012b] device. The device is similar to the LynxNet hardware prototype described above, but unlike it the CarMote is meant to be placed in a car. Compared to commercial off-the-shelf devices used for similar tasks (most typically smartphones), CarMote has lower production cost and lower energy consumption.

MansOS supports CarMote as a hardware platform and includes a number of applications implemented for it. Code listings in Chapter 3 (Listing 3.32 to Listing 3.35) show details on how MansOS can be applied to road surface monitoring. Listing 3.36 shows how MansOS can be used to program an application that uses accelerometer sensor to detect whether a vehicle is driving or standing in place (including standing, but with engine on – small vibrations are filtered out).

The latter application has also been implemented separately in MansOS for Zolertia Z1 hardware platform, using a more advanced algorithm based on Bayesian networks [Elsts 2014]. A Bayesian network is a form of probabilistic graphical model that describes a set of random variables and their probabilistic relationships. Bayesian networks are frequently used in areas where automated reasoning with uncertainty is required.

The algorithm in question used a pre-created Bayesian network that inferred the probability that the car is moving. In order to do that, the algorithm in turn used a set of classifiers with observable, dynamically changing values and their prior probabilities. The classifiers included examples such as whether the case was moving before and whether the readings of the accelerometer cross specific thresholds. The algorithm was tested on a test set that contained a number of everyday driving scenarios. The algorithm showed 100 % vehicle mode detection accuracy on these 31 scenarios (they included starting driving, starting in reverse gear, starting while turning, driving at various speeds, and coming to a halt). The algorithm positively compares to the algorithm previously developed by Mednis *et al.* [Mednis 2011] (which demonstrated only up to 80.6 % detection accuracy on the same test set).

An important point our experiments show is that it is possible to implement complicated probabilistic graphical models on very low-power MCU. The target MCU was MSP430F2617 (82 kb flash memory, 8 kb RAM) running at 16 MHz frequency. It required less than 30 milliseconds to execute the algorithm on this MCU. From this time, approximately 90 % was spent in calculating the classifier values, and only the remaining 10 % in the probability inference process itself. Such a per-

Table 2.6: WSN OS and Arduino feature comparison

	MansOS	TinyOS	Contiki	Mantis	LiteOS	Arduino
Multiple architectures	+	+	+	+	−	−
Preemptive multithreading	+	+[1]	−	+	+	−
Event-based programming	+	+	+[2]	−	−	−
App. development in C	+	−	+	+	+/−[3]	+/−[4]
Millisecond timing on TelosB	+	+/−	+/−	+/−	n/a	n/a

formance was more than satisfactory for our application. The system requirements required that the detection is performed with granularity not higher than 1 second; in order to achieve this granularity, the device is required to be powered no more than $30/1000 = 3\,\%$ of the time.

2.8 Concluding remarks

We have presented aspects of MansOS and compared it with state-of-art WSN operating systems and Arduino (Table 2.6).

MansOS demonstrates advances over other C-based WSN operating systems (Contiki, Mantis, LiteOS) in several areas: modularity, portability, and, to a lesser extent, reliability. It shows that it is possible to implement some of the ideas inherent in TinyOS in a different way, leading to an operating system that has better usability and is easier to learn.

We have identified several specific benefits present in MansOS. It provides:

- **Learnability.** The key is to provide abstractions that are either already familiar to the user, or require short time to grasp. For example, a Contiki programmer who wants to use software timers has to think in terms of hardware timer ticks. TinyOS programmer has to either do the same or to learn about binary milliseconds. In contrast, MansOS allows to use standard milliseconds as time units even on TelosB platform (Section 2.5.2). MansOS also uses POSIX API for several system modules rather than sensor network-specific abstractions, in order to better accommodate the existing programming knowledge of those target users who are experienced in programming, but novices in WSN. However, in some cases it turned out to be more productive to tweak the standard slightly because of efficiency reasons (Section 2.5.3). Superior usability of C-based operating system such as MansOS over a nesC-based operating system (TinyOS) for inexperienced (first-time) WSN users is evidenced by results in a small-scale user study.

[1]Through TOSThreads [Klues 2009]
[2]Beyond the protothread abstraction
[3]LiteC++ is used
[4]A subset of C++ is used

- **Modularity.** The hard challenge is to implement high usability while not giving up on WSN-specific design goals, in particular, resource and energy efficiency. We solve it by providing it a semi-automatic component selection mechanism (previously seen in C-based WSN operating systems only in much more limited implementations) that allows to modularize the system and to include only those components the application needs. One example of an modular, interchangeable component in MansOS is the kernel itself. Two versions of the core scheduler exists: one that uses preemptive threads, and one that uses only callback functions that react on specific events. For many applications the kernel can be switched without changing the C source code of the application. MansOS also does extensive compile-time and link-time optimizations to remove unused code, and implements some aspects of parametrized hardware interfaces, allowing to reduce binary code size and RAM usage.

- **Reliability.** While there is still a long way to go in making MansOS more reliable, it is already able to help the developer in several ways. Firstly, it provides a set of macros for debugging (such as the `ASSERT` macro and a macro for run-time stack overflow checking), which are included in the system code and enabled by default. Secondly, to avoid stack overflow, worst-case stack usage checking is integrated in the build process (Section 2.4.2) Thirdly, it uses simple and clear API. This reduces the chance that the developer is going to get lost in technical details, and helps to gain the developer a clear mental image of the services that the OS is offering. It also allows to write shorter application source code (Section 2.6.1). Fourthly, the simpler API does not include functionality that can be reliably implemented only in specific conditions, rather than on the typical WSN hardware. For example, TinyOS offers periodic timers, while MansOS does not, because non-realtime OS on top of current WSN hardware cannot guarantee execution of periodic timers with arbitrary, user specified frequency. (Instead, MansOS allows to periodically reschedule a timer.) Finally, the core system uses only static memory allocation.

- **Portability.** In contrast to their desktop counterparts, WSN hardware architectures and platforms come in great variety and are often specially adapted to concrete applications. Therefore portability is a critical requirement for WSN software. MansOS is portable and runs on several WSN hardware platforms (Section 2.2.2). Because of the architecture of the code, the amount of platform dependent code in MansOS is small (e.g. 629 lines of XM1000 platform-specific code), and porting to new platforms is fast.

Some of the results (component selection mechanism and four-layer code architecture) have been adapted to Contiki with good results (more than 12 % reduction in resource usage, and more than 17 % reduction in source code size, respectively).

While learnability and portability have been MansOS design goals since inception, the methods with which they have been approached have changed through time. For example, the emphasis on UNIX-like abstractions was gradually reduced, as several of the initial approaches were discarded because they showed insufficient efficiency in the context of sensor networks. All in all, efficiency has become a primary design goal (along with the two others mentioned above) in time. This is easy to understand, when the constraint of limited number of man-hours spent developing the system is taken into account. The number of features and especially the flexibility of MansOS is lower than that of major existing WSN operating systems (TinyOS and Contiki), since it has been developed for a shorter time and by fewer people. On the other hand, the reduced flexibility has in many cases lead to actual superiority of MansOS: simpler API, fewer application code lines, reduced binary code size and RAM usage, reduced energy consumption (Section 2.6) for simple applications. We believe that the typical sensor network developer for the typical WSN use case does not need the full flexibility of the hardware level exported to the level of an operating system; instead, she wants a system that is efficient and easy-to-learn (i.e. fast to start using). By cutting the corner cases, we managed to advance in the directions of efficiency and simplicity.

Compared to the state of art operating systems, MansOS often lacks advanced functionality. However, clear separation must be made between missing functionality that can be added without fundamental changes in OS architecture, and missing functionality that cannot. The former requires implementation-only level work; the second also conceptual changes. For an example of the former, TinyOS and Contiki implements more advanced networking protocols; however, these protocols can be adopted to MansOS if there is a need. For example, the IPv6 networking stack in Contiki can be compiled as library; the author has successfully linked this library to MansOS applications, essentially showing that IPv6 support is possible without system-level changes. More examples include: API for advanced ADC functionality, API for advanced hardware timer functionality, more device drivers (e.g. for real-time clocks, for liquid-crystal displays, for USB communication).

An example of the latter kind of work: a fully automatic mechanism of resource arbitration (not present in other C-based WSN operating systems as well). See also Section 2.5.4 for more examples of this kind.

However, system-level only support for application development is insufficient to reach the objectives of this dissertation. As long as there is the requirement that applications should be developed in C programming language, the application developer cannot be fully freed from low-level details. For example, knowledge is required of details such as pointer semantics, memory allocation issues, the way unaligned structure fields are padded by the compiler, and so on. This puts stringent prerequisites on the application developer. These prerequisites cannot be expected to be satisfied when application developers are without professional or educational background in programming. Therefore, the sensor network application development framework described in this thesis is extended with a high-level language: the topic of the next chapter.

Chapter 3

High-level programming language and tools

Contents

3.1 Introduction

As argued in the Introduction and Chapter 1, many of those who would benefit from wireless sensor networks are novice programmers or nonprogrammers. Due to their number, their needs are important enough to be taken into account when designing sensor network programming languages and tools.

However, up to this date no single novice-friendly programming toolkit, model or language has gained significant recognition both in WSN user and researcher communities.

There are several possible reasons. First, the early research had theoretical emphasis. From 28 WSN programming approaches reviewed in a recent survey [Mottola 2011] only approximately half have real hardware implementations, and only one has been used in a real-world deployment [Mottola 2011]. Therefore they are not validated against real-world requirements.

Second, many of the approaches require prerequisites that are not realistic for application domain experts. For example, [Newton 2007] [Mainland 2008] use idioms from functional programming, which might be hard to grasp for people without computer science background. Pleiades [Kothari 2007] extend C programming language for WSN macroprogramming, therefore putting C knowledge as a prerequisite. Snlog [Chu 2006] use Prolog-like programming paradigm that is confusing to all but seasoned computer science professionals (to repeat one more time, programming in Prolog "is notoriously difficult for novices" [Pane 1996]).

Third, many are either limited by design to only a particular kind of WSN applications (as [Madden 2005] and [Bai 2009]), or only solve a particular programming problem (as [Mottola 2006] and [Newton 2007]).

Inspired by the work done in novice programming research [Pane 1996] [Robins 2003] [Maloney 2008] [Neal 1989], we propose **Se**nsor **A**pplication Development **L**anguage (SEAL): a programming language and development environment for WSN application development.

Our contribution is to show that even a very restricted language with no loops or recursion can handle the requirements of sensor network applications well enough, enabling WSN application programming for people who do not want to learn numerous computer science concepts first. (There is a number of research articles that survey the current sensor network applications and describe their typical traits, for example, [Romer 2004] [García-Hernández 2007] [Strazdins 2013], or categorize them, for example, [Bai 2009].)

SEAL features declarative syntax that is specifically tuned for the requirements of a typical sensor network application and therefore allows to describe application logic in more concise and less complex way (Section 3.5.1) compared to alternatives. For example, SEAL applications transparently include control flow elements common to most of WSN applications, such as setting up event handlers and switching to low-power modes. This allows the programmer to focus on the application logic itself, which typically consists of reading, processing, and dissemination of sensor data generated by hardware (Fig. 3.1).

A SEAL application consists of component use cases and their parameters (Section 3.3). SEAL allows the domain expert to think in terms that either are already familiar to her, or have concrete real-world counterparts (such as physical sensors). For example, SEAL has support for numeric literals with custom suffixes, allowing the user to specify constants in physical units (such as degrees or luxes) rather than raw sensor readings.

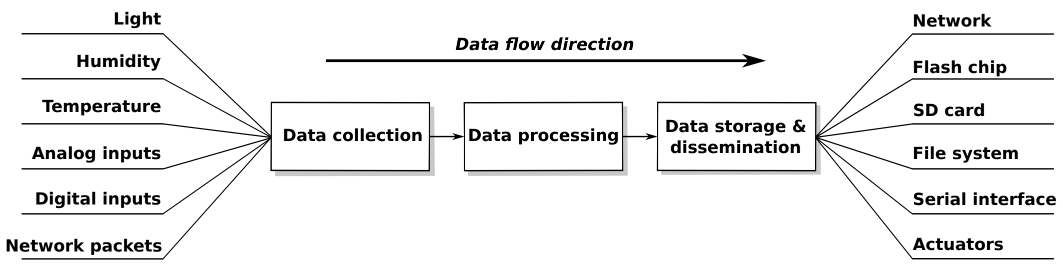


Figure 3.1: Typical data flow in a sensor node

SEAL code is compiled to C. For typical WSN applications SEAL runtime adds only a small or no overhead (Section 3.5.2). Therefore the code written by novice users can be used not only for application prototypes created during learning but also for real-world deployments.

This chapter uses materials from [Elsts 2012d] and [Elsts 2013a].

3.2 Our approach to design of a high-level sensor network programming language

3.2.1 Domain specific languages

As argued in section 1.3.5, domain-specific programming may be facilitated by offering the programmer either a domain-specific software library, embedded domain-specific language (eDSL), or a standalone domain-specific languages (DSL). The last approach has the highest implementation effort, but the lowest client effort of these three (i.e. the user has to do the least amount of learning).

It is generally believed that "programs written in a DSL also have one other important characteristic: *they can often be written by non-programmers*" [Hudak 1997] (emphasis ours). Using general purpose programming language such as C the novice user is confused by low-level details, such as pointer semantics and alignment of structure fields. Higher-level languages such as Java and Python are more forthcoming, but still require understanding of computer science concepts, for instance, loops and recursion. Another reason for choosing a DSL or an eDSL is that they both allow direct mapping between domain objects & processes and language elements. The result is the desired: abstractions with real-world counterparts.

One common problem from which DSL suffer is that of extensibility [Van Deursen 1997]. This is especially important because a lot of WSN applications use custom hardware [Strazdins 2013], which implies the need for application-specific software as well. Therefore the DSL should be designed in away that custom extensions can be added easily (without compiler modifications).

A DSL survey article [Van Deursen 2000] mentions a few other DSL benefits: their "programs are concise, self-documenting to a large extent, and can be reused for different purposes" and "DSLs enhance productivity, reliability, maintainability, and portability".

The same article also lists six *disadvantages* of DSL: (1) “the costs of designing, implementing and maintaining a DSL”, (2) “the limited availability of DSLs”, (3) “the costs of education for DSL users”, (4) “the difficulty of finding the proper scope for a DSL”, (5) “the difficulty of balancing between domain-specificity and general-purpose programming language constructs” and (6) “the potential loss of efficiency when compared with hand-coded software”.

In response to the first two points: by implementing the SEAL compiler and runtime, we showed that a DSL for WSN is feasible. The costs mentioned in the third point may be an issue, but they are insignificant compared with the costs of educating a domain-expert in a general purpose programming language. Scope (the fourth point) is indeed an issue, and is discussed in more detail in Section 3.2.2. The balance between programming constructs was tipped towards specific syntax, as it can be expected that the prospective users are not going to have extensive programming knowledge (Chapter 1), therefore using syntax that is similar to the syntax of some general purpose programming language would not help much. Finally, efficiency of applications programmed in SEAL is shown to be comparable to hand-written code (Section 3.5.2).

3.2.2 Separating user code from system code

Some aspects of WSN system programming are likely to remain complex. Often only application-specific hardware and software designs can achieve the necessary longevity of the system. Nevertheless, the *application logic* of real-world WSN applications is often trivial [Strazdins 2013]. Therefore, there is a need for clear demarcation line between system code and user code.

The *system code* is written by the WSN professional. The code describes internal mechanisms of WSN operating system (e.g. scheduling), device drivers (e.g. for sensors), and distributed algorithms (e.g. multihop routing).

The *user code* describes application logic for sense-and-send or event detection, as well as basic data processing, such as aggregation and filtering. It uses the system code to achieve its goals, but abstracts away from the low level details of the latter.

Therefore, a WSN programming approach should provide sandbox environment, where the users cannot do much harm for themselves, are not required to know computer science abstractions (such as classical data structures and algorithms), and are not confused by the too-many choices that are available.

3.2.3 Declarative or imperative?

Imperative languages (such as BASIC) are common for teaching novice programmers. In general, they allow to have clear mapping between the source code and the execution flow.

However, we argue that a declarative language is preferable in the context of our work. First, many WSN applications are event-based, which means that an imperative language will fail to reflect the execution flow as well! Therefore, the main strength of imperative languages is going to lose its importance in this domain. Rather, such a language would require explicit setup of event handling functions. If an imperative language and a polling-based approach is used, there are other issues: in this case a global execution loop is required, and low power mode semantics becomes explicit. A WSN-specific declarative language can hide all of the three: low level details of event handling, the global loop, and the usage of low-power modes.

Second, there is also the question of efficiency. In an imperative language, the behavior of a WSN node (e.g. the radio channel to use) can be set by modifying a variable or by changing some hardware state. This requires run-time overhead, code memory overhead, and, in the former case, RAM overhead as well. Declarative language allows to *declare* the behavior of the system. An advanced compiler can analyze this declaration and implement it in the compile time, avoiding any run-time overhead, except for hardware initialization. It is a well-known fact that for embedded software, efficiency is often more important than run-time flexibility!

At the same time, several typical WSN applications (for example, event detection based on sensor readings) are naturally expressed using finite state-machine abstraction, which is hard to implement in a declarative fashion. Therefore, a hybrid language that allows to have describe global system states may be expected to show the best results.

3.2.4 The design decisions of SEAL

- It should be a DSL with small grammar, and compact & intuitively readable syntax. Rationale: minimize client effort.
- Low level concepts should not be available (from the language). Rationale: limit the number of choices.
- Concepts with steep learning curves should be absent or not mandatory. Rationale: the novice user should be able to learn the toolset "on the fly".
- The expressiveness should be limited to application logic, not system logic. Rationale: see Section 3.2.2.
- Make commonly used patterns (the global loop, event handler setup, using low-power modes) implicit. Rationale: shorter, cleaner code. Studies show that novice users are confused by the semantics of low-power modes [Miller 2009].
- Make examples easy to access. Rationale: see [Neal 1989].
- Do as much as possible at compile time. Rationale: see Section 3.2.3.
- Hiding the control structures is a good idea. Rationale: the success of spreadsheets [Pane 1996], the success of SwissQM and WASP2 [Bai 2009].
- There should be an interactive IDE. Rationale: input aids provide hints and help to prevent syntax errors [Pane 1996].
- The IDE should be simple, not require learning more than 7–9 elements to program and test an application. Rationale: Miller's Law [Miller 1956].

- There should be a choice between textual and visual programming. Rationale: none of the two choices are superlative [Pane 1996].
- Usability evaluation and real-world use case evaluation should be present. Rationale: user behavior may be very different from designers expectations [Bai 2009].

3.3 Design and implementation

Consider the example application in Listing 3.1. The code reads temperature sensor with 10 second period and outputs the readings to the network. (Since no parameters are specified, the default collection tree protocol is used to transport the data to the base station.) It also monitors whether the temperature is above 40 degrees Celsius and turns the a LED on when this alarm condition is reached. Predefined suffixes, such as 's' and 'C' are used to specify physical units. The code generator automatically converts from these human-readable values to raw sensor readings. (Similar conversions cannot always be done with, for example, the C preprocessor, as it is not Turing-complete.)

Listing 3.1: Temperature monitoring application

```
read Temperature, period 10s;
output Network;
when Temperature > 40C:
    use Led, on;
end
```

3.3.1 Overview

Code written in SEAL is characterized by periodic execution flow. At the high level, the code is structured in *branches*: groups of statements. An *active* branch is a branch whose statements are executing (i.e. they are executed by the runtime, *periodically*; they may or may not be executing at a *concrete moment*). An inactive branch *starts* execution whenever all conditions that enclose it are satisfied. An active branch *finishes* execution either when all the statements in the branch have executed fixed number of times (as specified in parameters), or some of the conditions enclosing the branch are not satisfied anymore. *Subbranches* are created and controlled using specific statement types: `when` statements and `do` statements.

Table 3.1 shows language elements and their usage. The complete grammar of SEAL is given in Appendix A.

There are only two types of executable statements: component use cases and `set` statements. The former are executed periodically by default, the latter: just once. The rest of statements define or structure something.

There are three component types in SEAL: *sensors* (their "`use`" action is to read and return a value), *actuators* (their action: component-specific activation), and *outputs* (action: print, store or send sensor values, once they become available).

Table 3.1: Elements of SEAL

Element	Description	Templatized examples

Component use case	Specifies that the named component should be used (read or activated) in the current code branch, using the specific parameters supplied in parameter list. Starts with keyword `use` (can be replaced by `read` as a synonym for sensors, and by `output` as a synonym for system outputs), followed by the name of the component and list of parameters. The comma-separated list of parameters may include usage period, number of times to execute, and component-specific parameters.	`// light up the default LED`
		`use Led, period 1000ms, once;`
		`// "use" (i.e.  read) temperature sensor periodically`
		`read Temperature;`
		`// "use" (i.e.  output to) the network;`
		`// enable checksumming and select TDMA MAC protocol`
		`output Network, checksum, protocol TDMA;`

`when` statement	Determines which subbranches to execute depending on conditional expressions. A subbranch of a `when` statement is active whenever the condition that is associated with it (defined immediately after the opening `when`/`elsewhen` keyword) is satisfied, and no conditions associated with previously defined subbranches of this `when` statement are satisfied. There must be exactly one `when` subbranch, and can be zero or more `elsewhen` subbranches as well. The `else` subbranch has no associated condition. If present, it always must be the last.
	SEAL is flexible with regard to what can be used as a conditional expression. It can be a literal, a symbolic constant, system state, a (function of) sensor value, or any syntactically valid combination of the options, constructed using comparison operators and logical connectives. In the end, any nonzero integer value maps to Boolean *true*, zero – to *false*.
		`when <condition_1>:`
		`// executed when <condition_1> is true`
		`    use <component1>;`
		`elsewhen <condition_2>:`
		`// executed when <condition_1> is false`
		`// and <condition_2> true`
		`    use <component2>;`
		`else:`
		`// executed when both cond.  are false`
		`    use <component3>;`
		`end`

`do` statement	Determines the execution order of subbranches depending on their order in source code. The `do` subbranch that is first in the code becomes active immediately after the whole `do` statement itself starts executing. The subsequent `then` subbranches are optional; each of them becomes active after all the previous `do`/`then` subbranches have finished execution.	`do, once:`
		`    // executed just once`
		`    use <component1>;`
		`then, times 2:`
		`    // executed afterwards, twice`
		`    use <component2>;`
		`then:`
		`    // executed infinitely,`
		`    // after 1st & 2nd subbranch`
		`    use <component3>;`
		`end`

Table 3.2: Elements of SEAL (continued)

Element	Description	Templatized examples

`define` statement	Defines a new, "virtual" sensor as a function (possibly parametrized) of the currently defined sensors. After the definition, the new sensor can be used as any other sensor would be.	`// temperature maximum during application lifetime`
		`define MaxTemp max(Temperature);`
		`// a parametrized analog input`
		`define MyIn AnalogInput, port 2, pin 6;`

`set` statement	Initializes or changes a system state. A state can take any integer or Boolean values. Their value can be set from a literal, a sensor, a function, and a state (the same or another).	`// assign a constant`
		`set temperatureCritical False;`
		`// increment`
		`set counter add(counter, 1);`

`const` statement	Defines a symbolic constant. The name of the constant can be used in place of numerical literals.	`const MAX_TEMP 40C;`

`pattern` statement	Defines a named array of constants	`// "P" is the name of the array`
		`pattern P (100ms, 200, 300ms, 1s, 2s);`

`load` statement	Loads component library extension code (a Python file) or runtime extension code (a C file).	`// component definitions`
		`load "ExtensionLibrary.py";`
		`// runtime implementation`
		`load "ExtensionLibrary.c";`

`config` statement	Operating-system specific configuration, expressed in free-form string literal. In code generated for MansOS a single `config` statement corresponds to a single line in application's configuration file	`// Tell MansOS to disable networking`
		`config "USE_NET=n";`

Active outputs process values of all currently active sensors (unless specific subset of sensors is listed in parameters of the output's use case).

Wherever a numerical literal can be used, the same literal suffixed with symbolic suffix that carry type information can also be used. Futhermore, for some physical units the literal syntax is allowed to be put down in more complicated format as well; for example, a single time period can be equivalenty expressed in these multiple forms: `3785000`, `3785s`, `63min5s`, and `1h3min5s`. These dimensional values are converted to platform-specific sensor values or operating system-specific timer units in the compilation process.

SEAL is ultimately constrained by the fact that it is not Turing-complete. Predefined functions can be used and parametrized, but no new functions can be defined at the language level.

3.3.2 Semantics

In the design of the SEAL we had two conflicting requirements – we wanted to keep the syntax simple, and at the same time we wanted to provide expressive power. The solution was to delegate semantics to the level of individual components. In this way, behavior details are hidden in the user-configurable parameters of components. Additional expressive power comes from the unrestricted semantics of predefined data processing functions.

The benefits of this approach come from the fact that a single specific user is likely to use SEAL for just a single specific application (or for just a few specific applications), therefore she going to require relatively few of the components and data processing functions that the current SEAL runtime provides. Less effort is required to gain sufficient understanding of the behavior of specific, small subset of system-level components and data processing functions, compared to the effort required to learn and understand the more expressive and general syntactical constructions of other programming languages (such as loops, recursion, rich control structures such as the `switch/case` statement of C, pattern matching, etc.), their semantic counterparts, and their pragmatic implications.

In case the users of SEAL are given access only to simple components with straightforward semantics, the system should be suitable for teaching the basics of sensor network programming to complete novices. In case the users are allowed to access more complicated and powerful components (for example, components with a large number of static parameters, or components with dynamically adaptive behavior), the system is powerful enough for prototypes of real-world applications. Therefore the language can be adapted to different audiences without changing its syntax.

3.3.2.1 Sensing and actuating

Each run-time component to be used from SEAL applications must provide a C function for at least one action. The function should be provided as a name at the component library level, and as a definition (to use the C term) at the runtime library level. This default action is performed every time the component is "used" (i.e. a component use case is executed). For components that are sensors, this C function must *read* and return the value of this sensor. For components that are actuators, this function must *perform* some action; the return value of this function, if any, is ignored. (The third type of components, system outputs, will be discussed in the next subsection.) Additionally, components may have functions for turning them *on* and *off*. Sensors may optionally also have a function that allows to *act using a "sensor"*; for example, a digital input/output pin can be both read and written to, therefore its implementation-specific "use" function may be to write to this pin or to toggle its value. Some sensors also have a "pre-read" function, used to start the data acquisition operation on this sensor (see Section 3.3.3.4 for the motivation of such a function).

A single execution of a component use case (as defined in Table 3.1) corresponds to a single call of one of these functions. The exact function is dependent on the parameters of the use case; for example, parameter `on` tells to call the "turn on" function of the component rather than the default function. If parameters are contradictory, a compile-time warning is issued, and the run-time behavior is undefined.

The frequency of the execution of a component use case is also dependent on its parameters. The parameters that affect execution frequency and duration (*execution parameters*) are:

- `period` parameter tells to execute the use case periodically, using the time period specified.
- `duration` parameter tells to execute the use case no longer than for the time period specified.
- `times` parameter tells to execute it no more than the number of times specified (there are also shortcuts: `once` and `twice` parameters with no arguments).
- `on` and `off` parameters tell to execute it just once.

By default, a use case is executed as if it had a period parameter with value of one second.

All counters (duration, number of times already executed etc.) are reset once the subbranch in which the component use case is declared is reactivated after being inactive (because of a conditional expression that failed to hold).

More advanced control over execution timing is also available. Consider a user that wants to transmit a periodic, constant Morse-coded message using a LED. The desired on-and-off time intervals may be encoded as a list of constant numbers and put down in a SEAL application by using the `pattern` construct, which defines a named array of constants. The name of this pattern can then be used as a parameter to a component use case in order to perform the action of this component using a periodic pattern of timing. For example, Listing 3.2 shows how to transmit the SOS signal ("$\cdot\cdot\cdot - - - \cdot\cdot\cdot$" in Morse code) using the default LED. 100 millisecond time corresponds to both duration of a dot and a pause between dots; 300 millisecond time corresponds both to a dash and pause between letters; 700 millisecond – to pause at the end of the signal.

Listing 3.2: Using a LED to submit SOS signal in Morse code

```
pattern SOS (100, 100, 100, 100, 100, 300,
             300, 100, 300, 100, 300, 300,
             100, 100, 100, 100, 100, 700);
use Led, pattern SOS;
```

The idea of user controlled execution timing and intervals breaks down in the case of *hardware sensors* that themselves decide when to give data to the user, and in the case of sensors that are "read" from the *network*. However, in the first case, the sensors themselves signal about the new data (for example, by triggering an interrupt), and in the second case, library-level code makes sure that the SEAL application is notified about reception of new data packets (see Section 3.3.2.6 for details). These two kinds of sensors are described as self-reading in the documentation and the component library. Any attempt to read them by using explicitly specified execution parameters (`period`, `duration` or `times`) gives the user a compile time warning and has no effect on the run-time behavior.

SEAL applications can also include event-based logic. `when` conditions are used to match against specific runtime events; the execution of the component use cases included in the first subbranch of the `when` statement (e.g. reading a sensor or using an actuator) will be triggered by this event.

At the level of runtime-execution, polling-based event detection is always used for the typical sensors (the ones that are neither self-reading nor network-based). This means that these sensors cannot be referenced solely in `when` conditions; they must be explicitly read as well. For example, Listing 3.3 would not be a valid SEAL application with its first line removed.

Listing 3.3: Conditionally turning on LED

```
read Light;
when Light > 100lux:
  use Led, on;
end
```

The syntax in case of self-reading sensors or network-based sensors is the same as in Listing 3.3 (`read` statement should be present, execution parameters should be absent), but semantics are different. For these sensors, runtime execution may be made more efficient, because polling is not performed; instead, the logic associated with reading a sensor (such as the re-evaluation of conditional expressions) is triggered by other means (such as a hardware interrupt) when the new data is ready.

Finally, each sensor may also have meta-information to be read in addition to the value itself. Access to this information is supported through notation that is similar to the notation used to access object members in object-oriented programming languages. For example, the current implementation of the humidity sensor driver allows to read not only the value of the sensor, but also whether an error has occurred while trying to read the value. The value itself may be accessed either implicitly through `read Humidity` or explicitly through `read Humidity.value`; the error information: only explicitly, through `read Humidity.isError`. The component library describes which additional fields can be read for which sensors.

3.3.2.2 Outputting data

There can be multiple consumers of the sensor data. The most common ones are: the base station in the network, other nodes in the network, local storage devices, and local outputs (the serial interface, analog output pins, or attached actuators that accept incoming data values). All of these outputs can be accessed by using implementation-specific communication protocols; the protocols can range from very simple (e.g. for local outputs) to very complicated (e.g. dynamically adaptive routing and forwarding for multihop networks). The components can be used in various ways; for example, the network component can be used to send data either to the base station, to broadcast it, or to unicast it to some specific node in the network. Nevertheless, the protocol logic is encapsulated in runtime library components, and only high-level interfaces are exposed to SEAL via component library.

SEAL allows to use output components similarly to other components – i.e. under specific conditions, and using specific parameters. The action of a output component is to send out sensor values associated with this output. By default, all sensors read in an application are associated with each output. To limit the output to just a specific subset of sensors, the names of sensors in this subset should be enclosed as parameters to the output. For example, Listing 3.4 shows an application that reads both light and temperature sensors, but outputs just the light sensor readings to the network's base station.

Listing 3.4: Sending only light sensor readings to the base station

```
read Light; read Temperature;
output Network (Light);
```

Due to efficieny reasons, it is not always the best option to send out a stream of single sensor values. Therefore the sensors associated with an output are put in a packet, and send out all at once. This behavior can be reverted by specifying parameter `aggregate False` to the output use case – in this case sensor readings are sent one-by-one, as soon as they are read.

In case packets are used, a packet is sent out only when all the sensors that should be in this packet have been read. What if sensors are read using different time periods, but sending them out one-by-one is inefficient? There is also an option to include multiple readings of a single sensor within a packet. To do this, the number of the readings to include must be specified after the name of the sensor in output parameters (Listing 3.5).

Listing 3.5: Light sensor is read twice as often as temperature sensor and included in the network packet two times

```
read Light; read Temperature, period 2s;
output Network (Light 2, Temperature);
```

In order to efficiently determine which sensors already have been read and are present in the packet, a bitmask field is used. In the generated code, the bitmask is also included in the packet that is sent out, to allow the receiver to determine which sensors are present in the packet. (See Section 3.3.3.2 for more technical details.)

The field sizes of the packet can be determined by the code generator using information from component library. In general, the size of the field and its signedness should be compatible with the dynamic range of values for each sensor, which is specified in the component library by its designer (the default type of the field is

32-bit, signed integer). Structured data is not supported at the moment, and is not likely to be supported in the future. To allow the SEAL programmer to use e.g. three-axis accelerometer with three data channels, the component library designer either has to define three different components, or a single parametrized component that allows to read one of x, y, and z channels depending on which parameters are specified in SEAL code.

The outputs can be configured in application-specific way by specifying static parameters in SEAL code. The run-time behavior is implementation specific; the current implementation of the network component, which uses MansOS runtime, uses MansOS network sockets internally. The sockets provide an interface that allows both to send data to the network root node (the default behavior), to broadcast the data, and to send the data to a specific node described by its address. Of course, not all three options are supported by all MansOS routing protocols, but that is an implementation-level detail.

For example, the current component library code of the networking component allows to specify the following options:

- `protocol` – name of MAC protocol to use;
- `routing` – name of routing protocol to use;
- `destination` – specific destination address to use (use 0xffff to broadcast the packet; omit this parameter to send to the root node);
- `address` – Boolean parameter that tells whether to include the local address in data packets;
- `hopcount` – number of maximum hops a data packet can traverse in the network before being dropped.

The future work is also likely to add other parameters to the network component: physical layer parameters such as radio channel and radio output power; MAC layer parameters such as time-slot length and the maximum size of forwarding queue; routing layer parameters such as link selection policy, and so on.

These parameters allow to configure the network stack to suit application-specific needs. In case dynamically adaptive behavior is desired, it should be implemented at the level of underlying components. SEAL allows the developer to specify compile-time policies, rather than have the full flexibility of run-time changes.

SEAL also has syntactical sugar for working with files, in case a filesystem is implemented on local-storage devices (flash memory or SD card). Listing 3.6 shows how to output data to files: first, a binary file, which consists of a number of fixed-size records with structure similar to the structure used for network packets; second, an ASCII file, which consists of a number of lines in format "*sensor_name=sensor_value*".

Listing 3.6: Outputting sensor data to binary and text file

```
read Light;
output File, filename "lightData.bin", binary;
output File, filename "lightData.txt", text;
```

In contrast, Listing 3.7 shows how to output data from a file to the network. It includes a parameter named “where”, after which a conditional expression follows. This conditional expression is used to select a subset of the values in a file. In this example only light values larger than 100 luxes are sent to the network (i.e. the base station). In real world application, outputting from a file is not likely to be used in the way shown in the listing; it is more likely to be used in context of a `when` statement, with condition triggered by reception of some interactive “data query” command from the base station. An efficient implementation of query processing without building an on-node database is an area of future work.

Listing 3.7: Outputting data to a file and from a file

```
read Light;
output File, filename "Light.bin";
output Network, file "Light.bin", where Light > 100lux;
```

3.3.2.3 Processing data

In between reading and outputting data, the data may also need to be processed. For example, an application might want to calculate the average of multiple sensor readings and output only this average value in order to be more energy efficient. SEAL supports multiple pre-defined functions for data processing; it also allows to combine these functions to create derived functions.

Since SEAL has neither loops nor recursion, it cannot always do all of the required data processing on its own. However, in case more advanced processing is required, it can always be done at the system level, possibly implemented as an application-specific extension. On the other hand, if the processing is generic enough and can be implemented in efficient way without taking into account application-specific knowledge, it should be added as a predefined function at the compiler level. The SEAL compiler can be expected to slowly and naturally grow, as more applications are implemented in SEAL.

There are multiple families of predefined data processing functions:

- arithmetic;
- data agregation;
- filtering;
- signal processing;
- logical;
- special purpose and subset selection.

The functions, when applied to sensor values and constants, can be read just like any other sensors. They can also be used in conditional expressions.

Arithmetic functions include functions such as `plus` (addition), `minus` (subtraction), `multiply` (multiplication), `abs` (taking the absolute value), `neg` (negation), etc. For example, the value of the function `plus` is the sum of its two arguments. Each argument can be either a constants, a sensor, or a function of sensor values that returns a scalar value (as almost all functions do; the few exceptions belong to the "subset selection" family, and will be discussed later).

Listing 3.8: Using an arithmetic function

```
// return the sum of Light1 sensor and a constant
read plus(Light1, 13);
```

Data aggregation functions include `min` & `max` (taking minimum value and maximum value), `avg` (taking average value), `stdev` (taking standard deviation), etc. These functions have different semantics depending on their usage. If they are used as unary functions, they act on the stream of the past value of the sensor specified as their argument. If they are used as n-ary functions, they act on the *the latest* readings of their arguments (Listing 3.9).

Listing 3.9: Using a data aggregation function

```
// return the minimal value of Light1 sensor during the execution time of
    the program
read min(Light1);
// return the minimal value of the three light sensors at the moment
read min(Light1, Light2, Light3);
```

Filtering functions allow to reduce the frequency of sensor reading by outputting only readings that match specific conditions and suppressing the rest. They include `filterLess` & `filterMore` (allow to pass only values less/more than a specific threshold), `filterRange` (values in between two specific thresholds), `match` (allow to pass only values that match one of several constants), and `invertFilter` (invert the actions of all previously applied filters).

Listing 3.10: Using a filtering function

```
// return the values of the Light sensor that are below 100 luxes
read filterLess(Light, 100lux);
```

Signal processing functions may work either on a single value of a sensor reading, such as `map` function, which maps the sensor value from one dynamic range to a different dynamic range (the lower and upper bounds of the two ranges must be specified in parameters), or on the last n values (n should be specified as an argument) of sensor readings, such as the functions `sharpen` and `smoothen`, which use signal processing techniques to increase or decrease contrast between these multiple last readings. All these functions return a single value that corresponds to the latest value of the processed signal that can be calculated at the time they are called.

Listing 3.11: Using a signal processing function

```
// map the values of the Light sensor from 16-bit range (from 0 to 65535)
    to 10-bit range (from 0 to 1023).
read map(Light, 0, 65535, 0, 1023);
```

There is only one **logical function** in SEAL: `if` function. It allows to implement conditional data processing. `if` evaluates its first argument. If nonzero, it returns the value of its second argument; otherwise it returns its third argument. The third argument is optional; if not specified, and the first argument is zero, `if` function works the same way as a filter, by not allowing the values of the second argument to pass through.

Listing 3.12: Using the `if` function

```
// return the value of Light1 sensor if the random number generator
// returns nonzero; otherwise return the value of Light2 sensor
read if(Random, Light1, Light2);
```

Last but not least, there are **special-purpose functions**. However, before continuing with the description of these functions, it must be made clear that SEAL functions are allowed to be combined with each another. For example, SEAL can be used as a simple calculator by applying arithmetical functions on constants. The following line calculates and outputs the value of 13 + 25 * 4:

```
read plus(13, times(25, 4));
```

All of the functions discussed before can be combined with each another, not just arithmetical functions.

As for subset selection, the special-purpose function `take` allows to take multiple values of a single sensor. For example, it can be applied to calculate the minimal or average values – not during program's lifetime, but during the last *n* samples! There is also similar `takeRecent` function that allows to take values during the last *t* seconds, minutes, or hours.

Listing 3.13: Using an aggregate functions together with subset selection

```
// return the minimum of last six light sensor readings
read min(take(Light, 6));
// return the minimum of readings during the last 60 seconds
read min(takeRecent(Light, 60s));
```

As seen from the code example (Listing 3.13), `take` and `takeRecent` functions can be used as arguments to data aggregation functions, making the latter much more powerful. To clarify, a data aggregation function can actually take three types of parameters: the two types already discussed above, and a subset selection function. If, on the other hand, a subset selection function is used as an argument to any other SEAL function, compile-time error is generated.

The special-purpose function `sync` allows to read multiple sensors synchronously; its usage is described in Section 3.3.2.4.

In order to simplify the use of complicated functional expressions and allow shorter syntax, functions and their combinations, applied on specific sensor values or constants, can be *named*. In this way new, "virtual", sensors can be created. Listing 3.14 shows how to define a virtual sensor named `MinLight` that corresponds to the minimum of light sensor readings:

Listing 3.14: Defining a virtual sensor named *MinLight*

```
define MinLight min(Light);
```

The `MinLight` sensor can be used just like any other sensor, including use in conditional expressions. Furthermore, virtual sensors can also be parametrized. For example, the line:

```
define MyIn AnalogIn, channel 1;
```

defines an analog input named `MyIn` that by default reads the input channel #1.

3.3.2.4 Time-dependent algorithms

Temporal dependencies between actions can exist (for example, actions A and B, which both are triggered by the condition C, may always have to be performed in the order "first A, then B"). In the real world, the requirements for dependencies can frequently be more detailed than that; for example, "always perform action B exactly three seconds after action A is performed" may be a rule of application-level logic. Temporal actions between actions and data can exist as well: for example, activating an actuator may impact the environment, therefore changing the values of subsequent sensor readings.

One application example where time-dependent algorithm is required: a monitoring device attached to a larger system, observes the behavior of this system and tries to reset it in case "receiving high logic level from the digital input #1 tells that something is wrong; if this happens, try to reset the attached device by writing to the digital output #2, for three times with 2 second intervals; if this fails, start to blink the LED and give up". In order to implement this algorithm, the `do/then` syntactical construction is used (Listing 3.15). (Assume that using a digital output component means "set to high logic level for a short, system-dependent time and then release it". This logic could also be written down in SEAL, but is not done here, as that is not essential to the example.)

Listing 3.15: Using the `do` statement to achieve delayed action

```
define MyIn DigitalIn, port 1;
define MyOut DigitalOut, port 2;
read MyIn;
when MyIn == 1:
    do, times 3, period 2s:
       use MyOut;
    then:
       use Led;
    end
end
```

The whole `do` statement in this case is enclosed in `when` ... `end` brackets, therefore it starts and continues its execution only while the input remains high. In case the input falls low, the next reading of the `MyIn` "sensor" (performed periodically in the example, could also be interrupt-based) is going to force reevaluation of the `when` condition, and stop the execution of all of the logic inside the `do/then` statement.

As a side note, the `do` statement can also be used to group use cases that should be executed at approximately the same time. The benefit: code duplication is avoided, because paramaters such as `period` can now be specified only once (Listing 3.16).

Listing 3.16: Reading multiple sensors with the same period

```
do, period 2s:
    read Light; read Temperature; read Humidity;
end
```

In case a sensor in a `do` statement requires different execution parameters, it may override the values from the statement by specifying its own values.

However, even in the example above, fully synchronous reading of the multiple sensors is not guaranteed. In the current implementation, each sensor is still going to have a separate software timer associated with it. The timers are likely to fire one after each another, but this order may also be interrupted by a completely unrelated timer scheduled to fire at the same millisecond. As time passes, this may lead to sensor reading timers drifting away from each another, because at least in the current implementation, each timer is rescheduled in its own callback. In case sensors have to be read immediately one after each another, more strict guarantees are required. This can be achieved by using a special purpose function named `sync`. It takes sensor names as an arguments, and is guaranteed to read all of these sensors in a single timer callback.

Listing 3.17: Reading multiple sensors synchronously

```
read sync(Light, Temperature, Humidity), period 2s;
```

Returning to the `do` statement: it may be applied in case when multiple actions must be carried out in strict order, by putting them in different subbranches of the `do/then/then/.../end` syntactical construction. It can also be applied to avoid indeterminism caused by dependencies between sensors and actuators, e.g. by forcing always to read the sensor first and only then use the actuator which may affect the sensor readings.

As the reader can see, `do` statements essentially allow to enforce sequential execution flow – to simulate the behavior of imperative programming languages. For example, when a whole SEAL application consisting solely of non-periodic component use cases, line by line, is enclosed in the `do/then/then/.../end` syntactical construction, the execution flow essentially mimics that of an imperative programming language. Explicitly controlling execution flow is not required often in the sensor network application examples we have met, therefore such a control is an optional feature. The user may specify it if needed, but does not have to write additional code or think about this feature otherwise.

3.3.2.5 State-dependent algorithms

The examples so far show how to implement algorithms that are dependent on time and on the current physical state of the system, as determined by sensor readings. However, algorithms that are dependent on the *previous* physical state of the system may be required as well. Also, the underlying software components from the runtime library often have more state on their own, state that is not fully accessible by a single "read" function.

Consider an application that implements the functionality of a thermostat. It runs on a device that has temperature sensor and a software-controllable heater attached, and tries to keep the temperature close to 20 degrees Celsius. If the heater was turned on and off as soon the 20 degree threshold was crossed, it would lead to frequent oscillations, which may be undesirable for some reasons. Therefore let us assume that the heater should be turned on only when the temperature is significantly lower (say, 19 degrees) and turned off when it is significantly higher (say, 21 degrees). In order to implement such an algorithm with hysteresis (i.e. dependence on past environment), a SEAL application has to remember whether it is in the "heating" phase or in the "allowing to cool" phase. To do this, the `set` statement is used (Listing 3.18). It allows to implement state-machine like logic.

Listing 3.18: A heating system with hysteresis

```
set isHeating false; // initially not heating
when Temperature < 19C:
    set isHeating true;
end
when Temperature > 21C:
    set isHeating false;
end
when isHeating:
    use Heater, on;  // turn on heating
else:
    use Heater, off; // turn off heating
end
```

The underlying system-level (C code) variables and constants can also be read from SEAL. (Variables cannot be written from SEAL in order to keep the semantics simple; a component may still allow to configure its variables by exporting them as parameters through the component library).

To access these variables and constants, two pseudo-sensors were added to the component library. These pseudo-sensors are named "Variables" and "Constants", respectively, and can be used similarly to any other sensors, except that the field to read must always be explicitly specified (see note at the end of Section 3.3.2.1). The name of this field corresponds to the name of a constant or a variable in C header files of the system. For example, `Variables.localAddress` may be used to access the network address of the current node. By using a `when` statement with condition dependent on the value of this variable, the developer may create an application that is going to behave differently depending on the node on which it is running.

3.3.2.6 Distributed algorithms

In sensor network, decisions based on sensor data from multiple nodes sometimes must be made. Intra-network data aggregation from multiple nodes is also frequently required. Therefore support for acquiring and processing remote data must be present.

For this reason, *remote sensors* were added to SEAL. These sensors are recognized by their names. The name of a remote sensor always starts with `Remote`, after which name of a remotely available sensor follows. This remotely available sensor can be either a physical or a virtual sensor that is present as a component on some remote platform.

For example, `RemoteLight` and `RemoteTemperature` are both names of remote sensors. There are also a few special values that are not sensors, but can be prefixed with `Remote`, for example `RemoteCommand` and `RemoteAddress`. The names of remote sensors can also belong to virtual sensors, defined in SEAL application code. `RemoteCommand` corresponds to a command that is received from some remote device; `RemoteAddress` is the address of this remote device.

Unlike the local sensors, remote values cannot be read periodically, but must be processed whenever they become available, i.e. are received from the network. However, they can be queried periodically, for example, by sending a `RemoteCommand` with a specific value.

Listing 3.19: Data query processing (fragment)

```
when RemoteCommand == QUERY_DATA:
   // temperature data stream from past is unicast to the querying node
   read Temperature, file "Data.bin", output Network, destination
       RemoteAddress;
end
```

The communication idiom supported by SEAL can be described as “pushing” the data, rather than “pulling” it. A node can ask and receive data from an another, remote node, but only if the remote node is running an application that explicitly outputs the required data. Therefore, distinct client-side and server-side application logic is required. Even though this communication pattern can be implemented in a more efficient way than using the traditional client-server architecture (e.g. using the “publish/subscribe” messaging pattern), it still requires explicit server-side support. In contrast, WSN macroprogramming frameworks that implement data pulling, such as Pleiades [Kothari 2007], allow to implement e.g. access to remote variables without additional server-side code. The approach used by Pleiades allows much higher expressiveness: it allows to implement the street-parking demo [Kothari 2007] in a single application code listing, while SEAL requires at least two separate listings.

One idea how to extend SEAL is to add `input` keyword in symmetry with the currently existing `output`. The keyword would allow the local node to ask for remote data without remote-side support at the application-level. However, this approach also complicates the semantics and the implementation; the decision whether and how to implement requires careful cost-benefit analysis, and clarification of application-level requirements.

3.3.3 Implementation and runtime

SEAL parser, semantical analyzer, code generator and component library are all implemented in Python, with help of a Python module called PLY (Python Lex-Yacc [ply 2012]). The current code generator produces OS-specific C source code, which then can be compiled to platform-specific executable.

The first three elements (parser, semantical analyzer, and code generator) are within a single application; this application is called *the SEAL compiler* in this thesis. This name is justified by many precedents in research literature (such as "the nesC compiler", which translates code from another high-level language to C), and the fact that for the SEAL compiler, the C code generator could be relatively easily (without changing majority of source code) replaced by a bytecode generator, leading to a "true" compiler.

Applications written in SEAL relies on operating system services for runtime execution support (Fig. 3.2). The OS behind SEAL runtime at the moment is MansOS, and the C code generated by the SEAL compiler uses MansOS networking services, sensor drivers, permanent data storage library, and other components and services. However, there are no conceptual obstacles why another WSN OS could not be used, as long as implements at least software timers and a few of SEAL components. There is a proof-of-concept implementation of SEAL subset in Contiki [Elsts 2012d]. The rest of the discussion in this section is OS-agnostic, except where explicitly said otherwise; algorithm pseudocodes do not contain MansOS-specific details.

There are 121 regression test cases included in the distribution to ensure continuous quality of the parser and code generator. The implementation is publicly available [sea 2012] under the NewBSD license.

3.3.3.1 The component library

In order to generate OS-specific C source code from SEAL application, OS-specific library of components is required. Among other things, the library stores C code templates for reading sensors, activating actuators, using outputs and so on. The code templates are stored as ASCII strings; these templates contain OS-specific function calls. For example, if light sensor is read from SEAL code, the code generator searches for a component named "LightSensor", looks up its reading function (this function is named "readLight" and is without parameters in MansOS), and generates C code that calls this function.

In general, each component that is accessible from SEAL application syntax corresponds to a Python class in the component library. Each hardware platform on which SEAL runtime may be executed corresponds to a Python module. When loaded, this module instantiates all components that are available on that particular platform. In the design of the component library, class inheritance is heavily used in order to avoid code duplication between similar components (e.g. two versions of a light sensor); conversely, module inclusion is heavily used in order to avoid code duplication between similar platforms (e.g. TelosB and Zolertia Z1). For example,

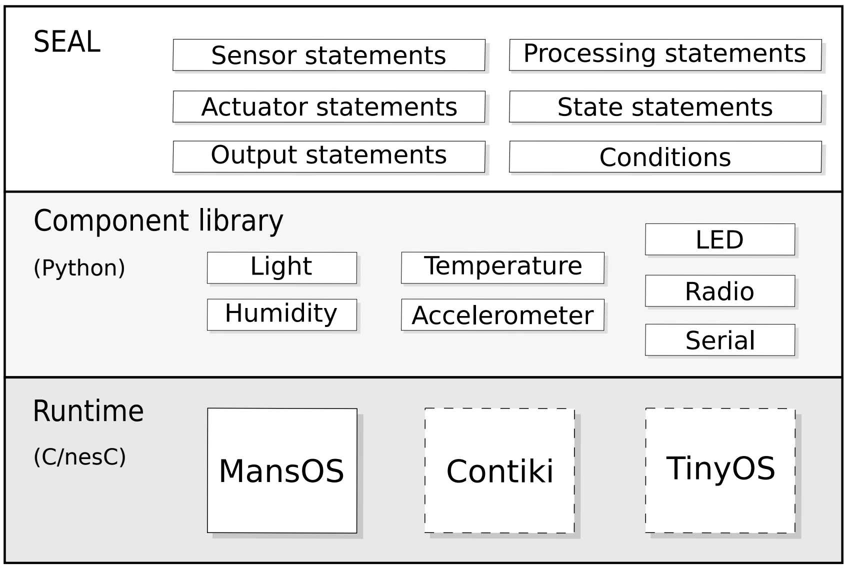


Figure 3.2: Elements of SEAL, SEAL component library, and the current and potential execution environments

there is a `SealSensor` base class that is common to all components that are sensors, and there is a `msp430.py` module that is included by `telosb.py` and other modules that use this MCU architecture.

The component library is designed to be easily extensible. For example, if a new component has to be added, only 5–10 additional lines of Python code are required. (For a concrete example, see Listing 3.28 and the discussion preceding it.) Furthermore, the new code is not required to be placed in the main component library; it can be put in an application-specific source file. Using `load` statement, the component may be made available from SEAL code. In this way SEAL can be be extended in application-specific way easily (without modifying the compiler or the default component library).

3.3.3.2 Runtime

The generated C code schedules software timers in order to read sensors and perform other actions periodically, and sets up hardware interrupt handlers to read interrupt-based sensors. Consider the example in Listing 3.1 again. Take just the first line: "`read Temperature, period 10s;`". If that would be the whole application, the corresponding generated code would be similar to pseudocode in Algorithm 3.1.

Now add the second line as well: "`output Network;`". This tells that the temperature sensor value should be not only read, but also used. The essential structure of the code generated from this updated applications is shown in Algorithm 3.2 (code repeated from Algorithm 3.1 is grayed out).

In the current implementation, a special kind of network packets is used to exchange information between nodes (see also Section 3.3.2.2). The details are

Algorithm 3.1 Pseudocode of temperature-reading application

```
Declare a global software timer named temperatureAlarm

function appMain( )
    Initialize temperatureAlarm with function pointer to temperatureCallback function
    temperatureCallback( )
end function

function temperatureCallback( )
                    ▷ The name of the function to call comes from the component library
    value ← readTemperature( )
    Schedule next temperatureAlarm callback after 10 seconds
end function
```

Algorithm 3.2 Pseudocode of temperature-reading application with output

```
Declare a global software timer named temperatureAlarm
Declare a global structure named networkPacket with temperature field

function appMain( )
    Initialize temperatureAlarm with function pointer to temperatureCallback function
    Initialize networkPacket with empty data fields
    temperatureCallback( )
end function

function temperatureCallback( )
    value ← readTemperature( )
    networkPacket.temperature ← value
    if all data fields of networkPacket are nonempty then networkPacketSend( )
    end if
    Schedule next temperatureAlarm callback after 10 seconds
end function

function networkPacketSend( )
                    ▷ The name of the function to call comes from the component library
    socketSend(networkPacket)
    Initialize networkPacket with empty fields
end function
```

not shown in the pseudocode, but the packet contains not only data, but also 4 byte header (2 byte field that signals the packet type, 2 byte checksum). After that, a bitmask describing the fields that are included in the packet follows. The bitmask is variable length, 4 bytes by default, but can be extended if necessary. The bitmask is used to determine which fields are empty and which are not in an efficient way. After the bitmask, data fields follow in the packet. Currently each field is always 4 byte long and contains a single sensor value. An area for future optimizations is to reduce these size requirements, especially since the component library already includes information about ranges of sensor values. The values used as sensor identifiers are compile time constants; they are allocated in application-specific way in order to reduce the length of the bitmask. The checksum field is filled in with checksum calculated on the rest of the packet, immediately before the packet is sent out.

SEAL code relies on the operating system for network communication. The maximal complexity of applications that can be implemented in SEAL are bounded by the capabilities of the runtime, which in turn is determined by the capabilities of the OS. The one extreme is application with no network communication at all, for *wired* sensor networks, or sensor-actuator "networks" where all communication is local to a single sensor node. The other extreme is a network that requires full point-to-point communication possibilities. In the latter case, these capabilities can be used from SEAL by specifying `destination` parameter for a network `output` statement, or by "reading" and checking the remote address of received data packets.

Now consider the whole application in Listing 3.1. The application contains not only data reading and output, but also conditional branching. The SEAL compiler translates it to the pseudocode in Algorithm 3.3 (code repeated from Algorithm 3.2 is grayed out).

The run-time state of each conditional expression (in SEAL syntax, conditional expressions are part of `when` statements) is stored in a Boolean variable. The state is reevaluated whenever it may change; e.g. when a sensor that is part of the conditional expression is read. This allows to implement event-based actions. Whenever a conditional expression changes its value, code subbranches associated with it are started or stopped.

Algorithm 3.3 Pseudocode of temperature monitoring application (Listing 3.1)

```
Declare a global software timer named $temperatureAlarm$
Declare a global software timer named $ledAlarm$
Declare a global structure named $networkPacket$ with $temperature$ field
Define $NUM\_BRANCHES$    1
Define $NUM\_CONDITIONS$    1
Declare a global Boolean array $branchActive$ with size $NUM\_BRANCHES$
Declare a global Boolean array $conditionsFulfilled$ with size $NUM\_CONDITION$

function APPMAIN( )
    Initialize $temperatureAlarm$ with function pointer to $temperatureCallback$ function
    Initialize $ledAlarm$ with function pointer to $ledCallback$ function
    Initialize $networkPacket$ with empty data fields
    TEMPERATURECALLBACK( )
                                    ▷ Similar code is generated for each non-default branch
    $branchActive[1] \leftarrow$ BRANCH1EVALUATE( )
    if $branchActive[1]$ then BRANCH1START( )
    end if
end function

function TEMPERATURECALLBACK( )
    $value \leftarrow$ READTEMPERATURE( )
    CONDITION1CALLBACK($value$)
    $networkPacket.temperature \leftarrow value$
    if all data fields of $networkPacket$ are nonempty then NETWORKPACKETSEND( )
    end if
    Schedule next $temperatureAlarm$ callback after 10 seconds
end function

function LEDCALLBACK( )
    Turn LED on
end function
```

Algorithm 3.4 Algorithm 3.3, continued

```
function networkPacketSend( )
    socketSend(networkPacket)
    Initialize $networkPacket$ with empty fields
end function

function branch1Evaluate( )
                                    ▷ The code generator has determined the relationship
                                                    ▷ between condition #1 and branch #1
    return $conditionsFulfilled[1] = True$
end function

function branch1Start( )
    ledCallback( )
end function

function branch1Stop( )
            ▷ This is code is optional for this application – an area for future optimization!
    Deactivate $ledAlarm$
end function

function condition1Callback(temperatureValue)
    $conditionsFulfilled[1] \leftarrow temperatureValue > 40C$

                        ▷ Similar code is generated for each branch dependent on this condition
    $oldActive \leftarrow branchActive[1]$
    $branchActive[1] \leftarrow$ branch1Evaluate( )

    if $oldActive \neq branchActive[1]$ then
        if $branchActive[1]$ then branch1Start( )
        else branch1Stop( )
        end if
    end if
end function
```

3.3.3.3 Cache

The sensors used in WSN have wide range of characteristics and usage patterns. Karl and Willig [Karl 2007] report that "for some of them [..] the power consumption can perhaps be ignored is comparison to other devices on a wireless node. [..] For others, in particular, active devices like sonar, power consumption can be quite considerable. [..] In addition, the sampling rate evidently is quite important."

Consider just two specific examples: Sensirion SHT75 temperature sensor [Sensiron 2011] and an analog accelerometer sensor. By WSN standards, it is quite expensive to read the digital SHT75 sensor: the data conversion consumes 1 mA current for up to 320 milliseconds. Frequent sampling is not suggested for another reason as well: reading frequency higher than 1 Hz leads to self-heating of the sensor, causing incorrect measurement results.

Reading the analog accelerometer sensor, on the other hand, is cheap: all that is required is to sample internal ADC of the microcontroller. Reading frequencies 100 Hz or more are common.

To use these sensors, application developer has to remember minimal sensor read periods for each, and other constraints (power consumption, self-heating). It becomes a burden the developers working memory.

Our solution is to use cache for sensor readings. In order to save RAM the cache is disabled by default and used only for those sensors that are read faster in the application than their maximal reading frequency. The maximal frequency is specified in description of each component. Using cache is transparent to the user by default, although may be controlled from SEAL applications by specifying `cache` parameter in a component use case (the component must be a sensor). The parameter takes a Boolean value and tells whether to use cache or not.

For example, take the one-line SEAL application "`read Temperature, cache;`". The logic of the C code generated from this application is similar to the logic described in Algorithm 3.1. The one difference is that `readTemperature()` is not called directly from `temperatureCallback()` function. Instead, `temperatureCallback()` calls function `cacheReadSensor`, passing application-specific ID of the sensor and pointer to `readTemperature` as parameters.

Algorithm 3.5 Caching sensor values

```
function CACHEREADSENSOR(code, readFunction, expireTime)
    $now \leftarrow$ GETTIME( )
    if $now > sensorCache[code].expireTime$ then
        $sensorCache[code].value \leftarrow$ READFUNCTION( )          ▷ Expired: update!
        $sensorCache[code].expireTime \leftarrow now + expireTime$
    end if
    return $sensorCache[code].value$
end function
```

The function `cacheReadSensor` (Algorithm 3.5) is implemented at system level. It uses an array named `sensorCache` statically allocated in RAM that stores most recently read sensor values, as well as the time when they were read. The function also takes parameter `expireTime` that describes (in milliseconds) how old sensor reading values are considered "fresh"; for example, if `expireTime` is 100 milliseconds, then all values read more recently than 100 milliseconds ago are not re-read from sensors, but returned straight from the `sensorCache` array.

The `expireTime` in code generated by the SEAL compiler is dependent on sensor-specific information described in the component library; by default this time is one second.

3.3.3.4 Synchronizing sensor readings

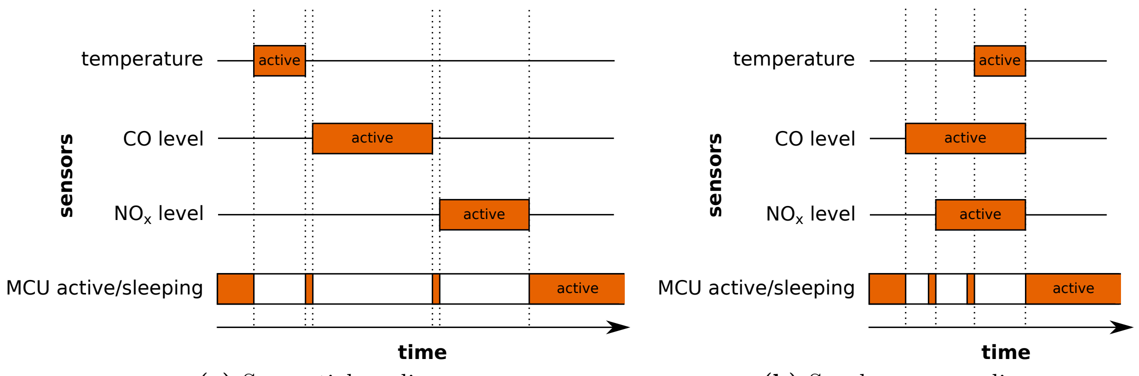
In order to perform a measurement, the SHT humidity sensor needs to collect and internally integrate data. Doing this requires significant time. Once the measurement is completed, its result is stored on an internal hardware buffer of the sensor chip. As a consequence, *reading* this result is much faster than *measuring* it. On typical sensor network hardware like TelosB, reading it can be completed in several microseconds.

Therefore, there is no need that the MCU of the sensor device remains active during the whole measurement sequence. In order to save energy, it can schedule measurement, go to low power mode ("sleep") while the measurement is performed, and wake up to read the result.

Gas sensors, such as for detection of Carbon monoxide (CO) or Nitrogen oxide (NO_x) levels require event more time to complete their measurements – up to several seconds. The same strategy should be applied to them.

Additional problems are created if the sensing device is mobile. For example, if the SHT humidity sensor is mounted on a car that is moving with speed 50 km/h, and the measurement takes 320 ms, the start and end locations of measurement are separated by 4.4 meters. This separation is going to increase in higher speeds and for sensors that require longer reading periods.

If multiple sensors are read sequentially (Fig. 3.3a, Algorithm 3.6), reading each sensor will take place at a different location. There is no reason to assume, by default, that this is what the user wants.



(a) Sequential reading **(b)** Synchronous reading

Figure 3.3: The behavior of two efficient algorithms for reading multiple sensors

Algorithm 3.6 Sequentially reading multiple sensors

```
Declare global software timers $temperatureAlarm$, $COsensorAlarm$, $NOxsensorAlarm$

function temperatureCallback( )
    $value \leftarrow$ readTemperature( )
    Schedule next $COsensorAlarm$ callback immediately
end function

function COsensorCallback( )
    $value \leftarrow$ readCOsensor( )
    Schedule next $NOxsensorAlarm$ callback immediately
end function

function NOxsensorCallback( )
    $value \leftarrow$ readNOxsensor( )
    Schedule next $temperatureAlarm$ callback after sensor read period
end function
```

Algorithm 3.7 Synchronously reading multiple sensors

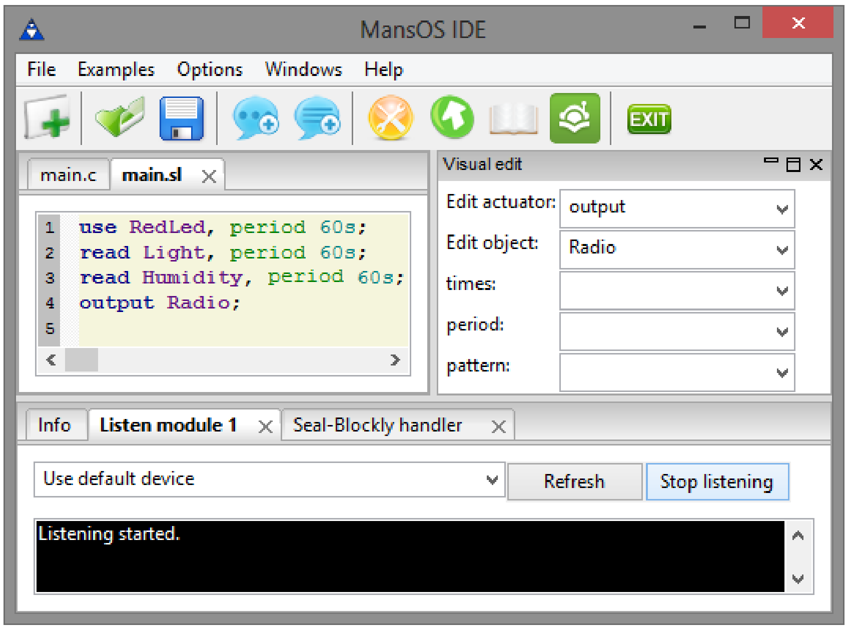
```
Declare global software timers $allSensorAlarm$, $temperatureAlarm$, $COsensorAlarm$,
$NOxsensorAlarm$

function allSensorCallback( )
    $value_{Temp} \leftarrow$ readTemperature( )
    $value_{CO} \leftarrow$ readCOsensor( )
    $value_{NOx} \leftarrow$ readNOxsensor( )
    $period \leftarrow$ sensor read period
    Schedule next $allSensorAlarm$ callback after $period$
    Schedule next $temperatureSensorAlarm$ callback after $period - READ\_TIME_{Temp}$
    Schedule next $COsensorAlarm$ callback after $period - READ\_TIME_{CO}$
    Schedule next $NOxsensorAlarm$ callback after $period - READ\_TIME_{NOx}$
end function

function TemperatureSensorCallback( )
    Start temperature measurement and return
end function

function COsensorCallback( )
    Start CO level measurement and return
end function

function NOxsensorCallback( )
    Start $NO_x$ level measurement and return
end function
```

Therefore SEAL translator may generate code that uses the synchronous sensor reading algorithm (Fig. 3.3b). To achieve similar results in manually coded C application, relatively complex state machine logic is needed, or an extra timer for each sub-sensor is required. (Algorithm 3.7 implements the latter; generating code that would use just no extra timer variables is an area for future optimization.) In contrast, SEAL allows to do the same (i.e. use the synchronous algorithm) with simply specifying `sync` function on multiple sensors. For example, `define AllSensors sync(SHT75, NOx, CO);` defines a virtual sensor `AllSensors` that simultaneously reads SHT75 and the two gas sensors from the example.

Synchronizing sensor readings also helps in case packets are used: the user can be sure that all fields in the packet are filled in at the same time, and sensor periods have not occasionally desynchronized due to timer jitter.

3.3.4 Integrated development environment

Two graphical interfaces have been built in order to help the application programmer programming in SEAL. First, there is SEAL integrated development environment (IDE; Fig. 3.4).



```
use RedLed, period 60s;
read Light, period 60s;
read Humidity, period 60s;
output Radio;
```

Edit actuator: output
Edit object: Radio
times:
period:
pattern:

Use default device

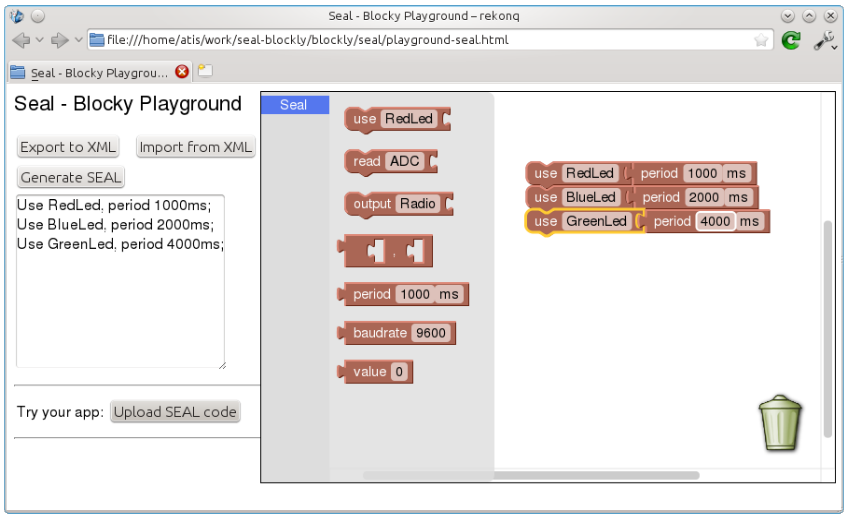
```
Listening started.
```

Figure 3.4: Editing SEAL application in IDE

The IDE can be used to enter code either by writing it on the left side of the window, or by selecting components and their parameters from dropboxes on the right side of the window. The latter approach allows to program by using only mouse clicks. As Fig. 3.4 shows, there are two types of elements on the edit window: dropboxes and checkboxes. The dropboxes with a few exceptions allows to enter integer parameters; the checkboxes: Boolean parameters. The form with the boxes is component-specific, and is automatically generated from the component's description in SEAL component library.

In the bottom of the screen, output window tabs are placed. One tab shows syntax errors and compilation messages. Another tab prints serial output of connected sensor nodes. The IDE allows single-click compilation and uploading from the toolbar. Last but not least, it simplifies access to examples: the IDE has a menu item that expands to a selection of example applications. The user can read their names and categories, and open their code with a single mouse click. The IDE was used throughout the second user study (Section 3.5.4).

3.3.5 SEAL-Blockly – sensor network visual programming


Seal - Blocky Playground – rekonq
file:///home/atis/work/seal-blockly/blockly/seal/playground-seal.html
Seal - Blocky Playgrou...
Seal - Blocky Playground
Export to XML
Import from XML
Generate SEAL
Seal
use RedLed
read ADC
output Radio
period 1000 ms
baudrate 9600
value 0
use RedLed period 1000 ms
use BlueLed period 2000 ms
use GreenLed period 4000 ms
Try your app:
Upload SEAL code

```
Use RedLed, period 1000ms;
Use BlueLed, period 2000ms;
Use GreenLed, period 4000ms;
```

Figure 3.5: Editing SEAL-Blockly application in Web browser

Then there also is SEAL-Blockly: a purely-visual interface to SEAL using Google Blockly [blo 2012] as a base. Scratch, a similar visual programming language [Maloney 2008] has been successfully used in teaching novice programmers. An interesting and important fact: using Scratch is so far removed from the usual perception how software development is performed that the users of the Maloney *et al.* research experiments failed to recognize their activities as programming!

SEAL-Blockly allows to create applications by putting puzzle-like blocks together and is usable from a web browser (Fig. 3.5). The basic idea is to translate each SEAL-Blockly block to a specific syntactic construction of SEAL language. The translated code is then automatically processed by SEAL IDE, which functions as a proxy between the hardware serial ports and the JavaScript executing in user's web browser. Eventually, the SEAL code is translated to C-code, which is the compiled in platform-specific way (Fig. 3.6).

Visual programming allows to avoid syntax errors and reduce working memory load, since all the possible blocks and their possible combinations are predefined. On the other hand, visual programming is not superior to textual in all contexts [Pane 1996], and visual programs can be harder to read [Green 1992], therefore we envision SEAL-Blockly only as one of *several* front-ends. Since SEAL code

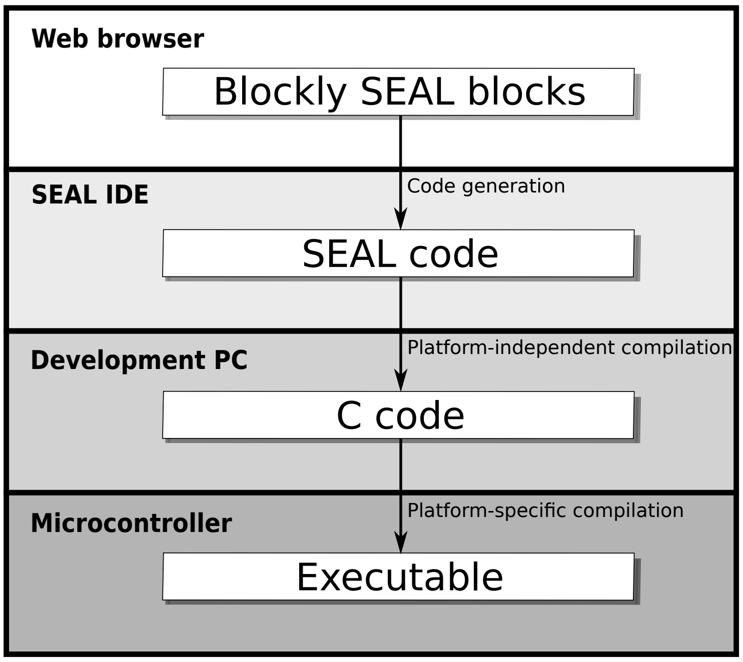


Figure 3.6: Software development workflow using SEAL / SEAL-Blockly

is automatically generated from SEAL-Blockly puzzle blocks, the user can switch between these two representations, choosing the one that she prefers.

3.4 SEAL in the context of related work

3.4.1 SEAL and the existing approaches

A partial relationship between SEAL and the related work (Chapter 1) is shown in Fig. 3.7. More complete overview of existing WSN programming approaches is given in Section 1.5; SEAL is compared with them in Table 1.4. Among the other approaches, there are few that take into account novice programming research. From these, SensorBASIC [Miller 2009] is an imperative language with large interpreter overhead. WASP [Bai 2009] is a language that is limited to a single application archetype.

TinyScript [Levis 2004a] is imperative language with event-driven execution mode that requires use of multiple source files even for simple applications.

`make`Sense framework [Casati 2012] is a new development with objectives similar to ours, but their target audience is different: users familiar with Business Process Modeling Language (BPML). Also, they use a three-level approach: the top level is a visual modeling language (an extension of the BPML), the medium level: a powerful custom Java-like macroprogramming language, the bottom: platform-specific executable code, in contrast to SEAL, which uses only two levels (as said before, the visual front-end SEAL-Blockly is roughly at the same level of abstraction).

The component model accessible from SEAL is conceptually close to SNACK

[Greenstein 2004], but the components in the component library of SEAL are higher-level; for example, “hardware timer” is a separate component accessible from SNACK. Also, SNACK uses advanced computer science concepts, is TinyOS-specific, and the authors did not perform usability study or validate their work in real deployments.

Multiple integrated, higher-level programming or data querying interfaces for sensor networks exists, TinyDB [Madden 2005] being the most prominent, but they lack the generality we are looking for. Similar arguments stand against sensor network programming using Web service architectures. (At least the currently existing.) For example, TinySOA [Avilés-López 2009] is too limiting as it offers no support for in-network data processing.

Sensor networks can be programmed using general purpose high-level languages such as Java [Smith 2007] or Python [pys 2013], but none of these options have gained high acceptance. First, they are heavyweight; second, unsuitable for novice programmers, as they presuppose existing programming language knowledge.

Embedded software programming languages (rather than formal specification languages or modeling languages) that try to do as much as possible at compile time is not as well explored research direction, although in author’s opinion, a promising one. One example of such a language is Virgil, which enables building of complex data structures at compile time, and is able to execute the binary “directly on the bare hardware without a virtual machine or any language runtime” [Titzer 2006]. However, Virgil is meant for resource-constrained embedded systems in general rather than sensor networks. Much can be done at compile time using the C preprocessor; however, it is not Turing-complete (since recursion is not possible), the syntax is not straightforward (especially for debugging), and the behavior is not always intuitive (leading to bugs when, for example, macro parameters such as `i++` are unexpectedly evaluated multiple times).

As for formal specification languages or modeling languages as such, they may require prerequisite knowledge, and invariably add one more layer of abstraction, which increases the time-to-learn and may increase the total complexity of the system (Section 1.3.6).

In our opinion, the problem is to choose the right tradeoff between being *too restrictive* (like TinyDB and WASP) and being *too allowing* – so allowing that users tend to get confused. The objective of SEAL is to allow to wide range of applications (Section 3.5.3.1), while purposefully keeping the complex parts of the application out of the reach of a SEAL programmer (Section 3.2.2).

In contrast to most of the models overviewed in Section 1.5, SEAL comes with an integrated development environment. Experimental results from [Bai 2009] and the success of Arduino both show the importance of IDE.

Whether visual programming that exceeds simple code wizards and helpers is needed – that, however, is a contentious issue. Research shows that visual programming is not better in all contexts, for example, visual programs may be harder to read [Green 1992]. Nevertheless, the members of author’s research group have implemented SEAL-Blockly: a visual sensor network programming environment. It is available as one of the two front-ends to SEAL. Therefore the novice can read example programs using textual interface, but modify or extend the program using visual tools, enjoying the best of both worlds.

3.4.2 SEAL and macroprogramming

There are two very different ways how future evolution of SEAL may accomodate existing work in WSN macroprogramming and distributed processing abstractions.

The first is to create or adopt a macroprogramming framework to generate SEAL code (i.e. macroprogramming on top of SEAL). The second way is to allow to access macroprogramming constructs from SEAL (i.e. SEAL on top of macroprogramming).

The first way was apparently envisioned by the authors of SNACK, who claim that "higher level mechanisms [on top of SNACK] are natural and probably inevitable". The second way was taken by *makeSense* team, where macroprogramming code is generated from BPML models created by the users of the system.

Properly implemented *macroprogramming on top of SEAL* would make possible implementations of SEAL code migration from node to node, adaptive self-optimization of SEAL applications, and complex distributed processing algorithms that require e.g. write access of remote variables. These programming models proposed by the WSN community are so far not considered in our work. (We note that they are absent in mainstream WSN programming as well, and their applicability to real-world WSN is yet a question.) Such an extension would provide higher-level view of WSN. However, the exact requirements of it in the context of novice users are far from being clear.

SEAL on top of macroprogramming is far more straightforward to implement and use. Leveraging on existing macroprogramming ideas, the functionality of SEAL applications would be extended. Additional runtime components could offer support for e.g. iteration over not only physical, but also logical neighborhoods (using e.g. [Welsh 2004] and [Mottola 2006]), serialized access to variables on neighboring nodes, distributed locking and distributed resource arbitration. However, this approach would be limited by the expressivity of SEAL syntax; additional changes may be required in it in order to fully accomodate the power of distributed algorithm in SEAL; for example, the addition of `input` keyword (see Section 3.3.2.6).

We have evaluated the adoption of existing frameworks. Pleiades [Kothari 2007] is a reasonable choice, because it is an extension of C language, so it is straightforward to adopt the SEAL compiler to generate Pleiades code. There could be a specific network component accesible from SEAL for e.g. searching a value in network that would internally use Pleiades `cfor` loop. This specific type of `for` loop provides support for serialization, distributed locking, and `break` functionality. The logic of the `for` loop would be transparent to the SEAL user; he would be see only the result of the search operationg, likely as a stream of values presented as a specific sensor-type component.

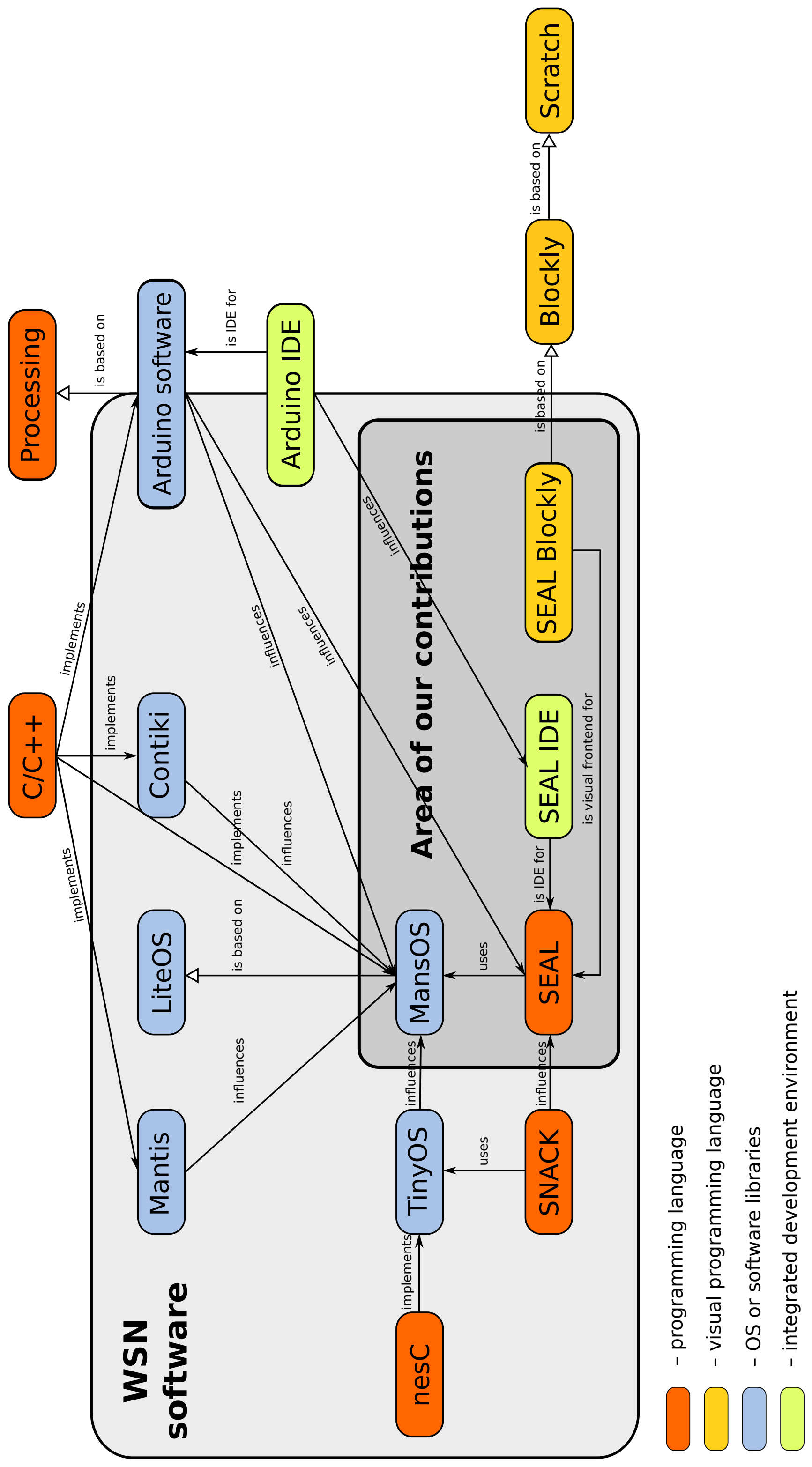


Figure 3.7: SEAL in the context of related work

3.5 Evaluation

3.5.1 Analytical evaluation

The usability of SEAL can be shown analytically in two ways:

1. by comparing application metrics (source code length, cyclomatic complexity, and programmer's effort according to Halstead complexity measures);
2. by comparing the complexity of the syntax of the language.

The code generated by the SEAL compiler puts the system in low power mode by default. In contrast, when programming in a language like SensorBASIC with its single-flow imperative execution model is used, the application developer is required to include explicit `sleep` commands. Studies have shown [Miller 2009] that novice users do not have clear grasp of low-power mode semantics and fail to use the `sleep` keyword correctly.

Compared with programmers using languages with explicit event-based execution flow (like TinyScript), SEAL developer does not have to think in terms of event handlers, and implicit state machines, which are arguably hard for novices to grasp.

3.5.1.1 Syntax complexity comparison

The language is small compared even to an already parsimonious general-purpose language like C (Table 3.3). It has shorter grammar, fewer keywords, fewer operators (only arithmetical and comparison), and smaller character set, therefore the cognitive effort required to learn the syntax of the language is reduced. The complexity of SEAL syntax is comparable to TinyScript and nesC, although the user of nesC has to know C as well.

Table 3.3: Metrics of several sensor network programming languages: SEAL, C [c-i 1999], nesC [Gay 2009], and TinyScript [Levis 2004a]

	SEAL	**TinyScript**	**C**	**nesC**
Nonterminal symbols	**36**	31	70	24 new + 13 changed
Keywords	**19**	15	31	15 new
Operators	**8**	23	46	1 new
Non-alphanumeric characters with special meaning	**14**	19	30	0 new

There are 19 *keywords* in SEAL:
`and`, `config`, `const`, `define`, `do`, `else`, `elsewhen`, `end`, `load`, `not`, `pattern`, `read`, `or`, `output`, `set`, `then`, `use`, `when`, `where`.

The number is smaller than the number of keywords in C (31 keywords). As for WSN DSL: nesC has 15 additional keywords [Gay 2009] to all of C keywords. The usage *scope* of keywords is actually greater in SEAL: C includes specific characters (such as `&` and `|`) for logical operations while in SEAL the operations require keywords. Therefore the number of *operators* can be greatly reduced: 8 in SEAL *vs.* 46 is C. Operators in SEAL are used only for comparison.

The C keywords are (31 total):
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

Additional nesc keywords (15 total):
as, call, command, components, configuration, event, implementation, interface, module, post, provides, signal, task, uses, includes.

SEAL *grammar* is also quite short: as specified in Appendix A, it has only 36 nonterminals.

For comparison, C grammar (C9x ISO draft) [c-i 1999] has 70 nonterminals.

We hypothesize that a large number of non-alphanumeric characters is confusing for novice users, because it increases the unfamiliarity of the notation. This is especially true with regard to characters other than the well-known grammatical symbols (e.g. comma) and arithmetical operators. To put it simply, large number of strange characters makes the code less readable for novices. It is harder to intuitively grasp the intent of unfamiliar code that is full of symbols whose meaning is not recognizable from everyday contexts, compared to unfamiliar code that uses plain English and a few well-known characters.

Also, some of the characters used in C-like syntax may be unfamiliar to the user as such. For example, when teaching how to perform a task (unrelated to SEAL) to users who belonged to the potential target audience, we recorded a case when a user failed to find the ’|’ character on keyboard.

SEAL keeps the number of non-alphanumeric characters small: only the following are present in the language: ’.’, ’,’, ’:’, ’;’, ’(’, ’)’, ’<’, ’>’, ’+’, ’-’, ’!’, ’=’, ’"’, ’_’.

Additionally, two ’/’ signal the start of comments.

In contrast, C has additional characters [c-i 1999]:

’{’, ’}’, ’[’, ’]’, ’~’, ’^’, ’/’, ’*’, ’?’, ’%’, ’’’, ’&’, ’|’, ’#’, ’\’.

SEAL has 16 extra characters in total, while C has almost twice as much: 30 (excluding well-known arithmetical operators: 14 and 26 respectively). In typical applications, the non-alphanumeric characters are also used with higher frequency in C than in SEAL.

Last but not least, SEAL syntax is insensitive to case, helping to avoid subtle syntax errors [Pane 1996].

In sum, SEAL has simpler syntax. Even though syntax is not the sole component of a language, it is still an integral and necessary part, required to use the language confidently. Therefore simpler syntax should lead to a language that is easier to learn.

3.5.1.2 Application source code metrics

Source code metric comparison. Initially, four test applications were implemented:

- *Empty* – an application with no user logic;
- *Sense & print* – periodically sample sensors (light, humidity, and temperature) and send results to serial port;
- *Sense & send* – same as *Sense & print*, except that sensor values are sent to network and stored in external flash memory;
- *SAD* – a real-world application for environmental monitoring, conceptually similar to the third application (Chapter 4).

For comparison, the first three applications were also implemented in nesC, plain C (using MansOS API), TinyScript, and SensorBASIC. SAD was implemented only in SEAL and in plain C, as TinyOS lacked certain specific components. The MansOS C implementation adhered to MansOS coding guidelines; TinyOS implementation used the same coding style as in TinyOS application examples.

Afterwards, a second batch of applications was implemented, as a prerequisite for the user evaluation study (Section 3.5.4). SensorBASIC implementations for these were taken from [Miller 2009]. We were unable to test any of SensorBASIC implementations because the interpreter is not publicly available.

We used three metrics: number of lines of code, cyclomatic complexity, and programmer's effort according to Halstead complexity measures [Halstead 1977].

Cyclomatic complexity is essentially the measure of the branches in the control flow of a program [Jones 1994]. High cyclomatic complexity implies convoluted, hard-to-understand branching. This metric shows good results even when comparisons are done across multiple programming languages.

Line count also gives some suggestions how easily the code can be understood, especially if the languages have similar levels of abstraction. It was evaluated excluding comments and empty lines, but including MansOS configuration files.

The last metric is a software complexity measure based on the numbers of operators and operands in an application. Operator is something that carries out an action; operand is something that participates in an action. According by Halstead, the effort required to create an application is directly proportional both to its volume and difficulty. The difficulty measure suggests how difficult it is to develop or understand the code of the application. Halstead metrics identifies new operators and repeated operands as the he sources of difficulty in an application.

More formally, for a given problem, let [Halstead 1977]:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, derived measures are calculated:

Table 3.4: Cyclomatic complexity comparison

	SEAL	**BASIC**	**TinyScript**	**C**	**nesC**
Empty	**0**	0	0	1	1
Sense & print	**1**	1	2	2	5
Sense & send	**1**	1	2	2	23
SAD	**2**	n/a	n/a	4	n/a
Exercise 1	**1**	1	4	2	2
Exercise 2	**2**	2	4	4	5
Exercise 3	**3**	3	4	4	5
Exercise 4	**1**	3	3	4	6

Table 3.5: Lines of code comparison

	SEAL	**BASIC**	**TinyScript**	**C**	**nesC**
Empty	**0**	0	0	3	6
Sense & print	**4**	6	6	15	49
Sense & send	**5**	9	7	42	179
SAD	**14**	n/a	n/a	185	n/a
Exercise 1	**1**	5	8	8	24
Exercise 2	**4**	4	8	13	48
Exercise 3	**5**	5	11	14	34
Exercise 4	**4**	11	16	22	56

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

According to the original Halstead's interpretation, the "effort" was directly proportional to the time required to developed the application; he provided approximation for time T: $T = \frac{E}{18}$ seconds. However, this interpretation cannot be simply extrapolated to domain-specific declarative languages without additional research.

The Halstead complexity metric was not calculated for nesC implementations (except for the first application, in order to show that it much higher than in high-level languages), and also not for the implementation of SAD application in C, as exact calculation of these metrics would be to time-consuming. It is clear that compared to SEAL, the complexity would be higher by several orders of magnitude. The empty application was not included in the comparison of Halstead complexity, as the formula for calculating Halstead complexity breaks down in case the number of operands is zero.

SEAL component use case parameters were counted as operators in Halstead complexity measures. External function calls were also counted as operators (except

Table 3.6: Effort comparison according to Halstead complexity measures

	SEAL	**BASIC**	**TinyScript**	**C**	**nesC**
Sense & print	**134.8**	67.5	1617.5	2443.6	1 310 048.9
Sense & send	**180.0**	199.7	2244.0	32 352.2	n/a
SAD	**5725.8**	n/a	n/a	n/a	n/a
Exercise 1	**31.0**	183.6	1353.7	495.0	n/a
Exercise 2	**421.9**	561.2	1294.9	1832.0	n/a
Exercise 3	**419.4**	552.0	2871.9	3664.6	n/a
Exercise 4	**917.4**	7625.3	14 592.4	17 849.7	n/a

sensor reading function calls, in order not to penalize languages that use implementations with a separate "read" function for each sensor). In contrast, function definitions were counted as operands.

Results and discussion. The results are given in Table 3.4, Table 3.5, and Table 3.6. For applications with no source code the cyclomatic complexity of the applications code was evaluated as zero.

As expected, SEAL source code sizes are much smaller (at least 4 times, more for complicated applications) than those of C and nesC (which is especially verbose).

Languages that make event-handing explicit (TinyScript and nesC) unsurprisingly have noticeably higher cyclomatic complexity.

TinyScript and BASIC source line count is similar to SEAL. However, they are more complex, especially TinyScript, which utilizes several additional concepts: variable declarations, array element access, data types, and explicit usage of timers.

SEAL showed lower programmer's effort according to Halstead complexity measures compared to BASIC, TinyScript, and C in all except one cases: for the very simple "Sense & print" application, BASIC performed better. The outlier case can be explained by the fact that BASIC uses the newline character as an implicit separator between statements, while SEAL uses the ";" character as an explicit separator. The newlines were not counted as operators in the evaluation; the semicolons were; therefore the number of operators in this SEAL application was higher. Interestingly, the complexity of implementations in plain C comparable to those in TinyScript, with just one exception.

We admit that the differences between SEAL and BASIC for the very simple applications are inconclusive; in fact, BASIC achieves great clarity and simplicity of code. However, the advantages of SEAL become visible as the applications get a little more complicated. The contrast is especially high for "Exercise 4" implementations. The SEAL implementation was shorter because it used primitives provided by the language to calculate absolute and average values of sensor readings; in the other languages, this functionality had to implemented in the application itself.

In whole this evaluation, which includes comparison of three metrics on 34 implementations of 8 applications in 5 languages, *there is only one case in which SEAL scores lower than any of the competitors in either of the metrics.*

Table 3.7: Code memory and RAM usage overhead of SEAL applications

	Code memory		RAM	
	TelosB	**SADmote**	**TelosB**	**SADmote**
Empty	75.8 %	26.0 %	7.0 %	2.1 %
Sense & print	21.1 %	7.1 %	29.9 %	15.0 %
Sense & send	23.8 %	16.7 %	39.1 %	9.8 %
SAD	n/a	11.6 %	n/a	4.8 %

Data processing support. For data processing SEAL offers several dozen built-in functions. The application in Listing 3.20 demonstrates their usage: it calculates the absolute difference of two light sensor readings, averages last 10 readings, and outputs the result. The code is understandable by nonprogrammers (Section 3.5.4). Equivalent TinyScript and SensorBASIC applications are 10–20 lines long and contain much more details.

Listing 3.20: Calculating the average difference

```
define Diff difference(TotalSolarRadiation, PhotosyntheticRadiation);
define AverageDifference average(take(Diff, 10));
read AverageDifference, period 500ms;
output Serial;
```

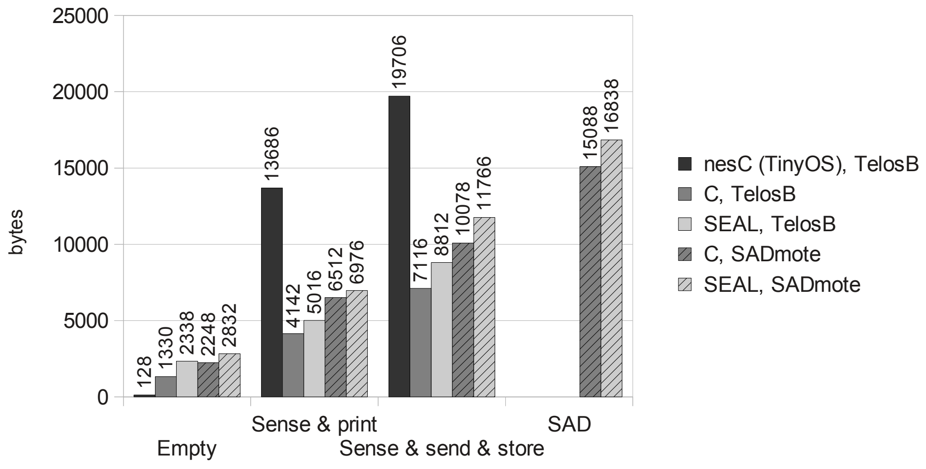
Network-programming support. The SEAL compiler also facilitates WSN programming by automatically generating code for different roles in the network. In particular, SAD application requires the user to write the code not only for the mote, but also for the base station role and two data-forwarder roles. The code is short, but has to be placed in 6 additional files. The SEAL compiler generates source code for each of these roles automatically, after the user has named the desired data collection protocol and its parameters in application's code.

3.5.2 Efficiency and feasibility evaluation

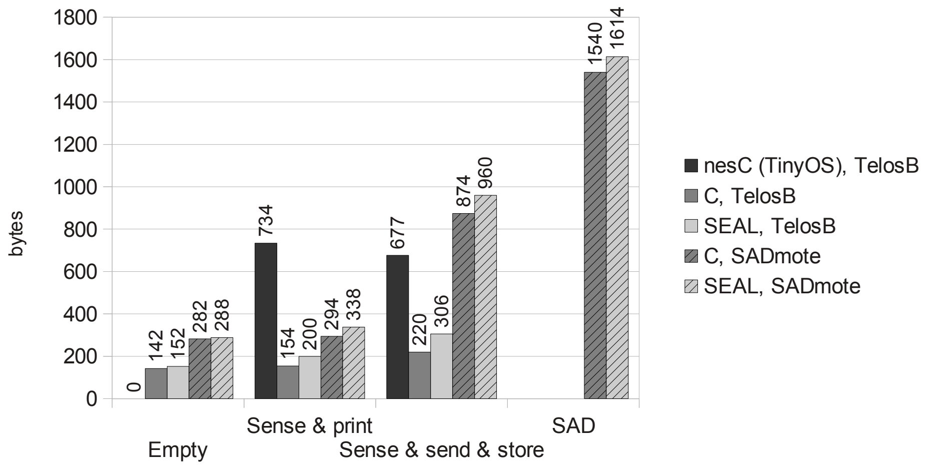
We measured binary code size and static RAM usage of the test applications for TelosB and SADmote platforms (SADmote is a custom MSP430-based hardware platform used in SAD application, see Chapter 4). The versions of the operating systems used were the same as for MansOS evaluation (Section 2.6).

Compared with native implementations in MansOS, applications generated by the SEAL compiles have only 7–24 % code memory overhead and 2–40 % RAM usage overhead, excluding the empty application (Fig. 3.8, Table 3.7). Compared with TinyOS on TelosB, SEAL applications actually demonstrate better results. One reason lies in the declarative nature of SEAL: the programmer is allowed to declare system's behavior and policies at *compile-time*, avoiding any run-time overhead. In sum, our implementation of SEAL is certainly feasible on very low-power microcontrollers.

MansOS has support for partial runtime reprogramming (see Section 2.5.1.3). Specifically, user code (application logic) is distinguished from system code (OS services) and can be reprogrammed separately. Therefore it is profitable to reduce



(a) Code memory usage



(b) RAM usage

Figure 3.8: Application binary code size and RAM usage comparison of SEAL applications

the size of the user code. The user code portion sizes in example applications (implemented in SEAL) are: 384 bytes for *Sense & print*, 2212 bytes for *Sense & send & store*, and 4454 bytes for SAD (all evaluated on SADmote platform), making partial reprogramming several times less expensive.

Energy usage was not evaluated separately for SEAL applications (see Section 2.6.5). While energy usage as such is extremely important in WSN context, SEAL adds few new aspects to the problem. SEAL applications are compiled to C code, which uses execution model comparable to that of hand-written MansOS applications. The execution model at the lowest level uses: software timers for all periodic operations, interrupts for all event-based operations, and spends the rest of the time in the lowest-power mode that is safe to use. Therefore the energy usage footprint of node-local SEAL applications is also comparable to hand-written applications.

The situation is more complicated with regard to network communications. As the current implementation is on top of MansOS, it does not provide support for multihop unicast communication in the network (except to and from the base station). Efficient implementation of such a communication is a future challenge. For other forms of communication, there is some overhead associated with the extra fields included in the packets used for communication: packet type (2 bytes), and a bitmask of the sensors included in the packet (at least 4 bytes). The sensors at the moment also are packed inefficiently: a 4 byte field is always used for a single sensor reading, even though the dynamic range of the sensor might be much smaller. Finally, there is no doubt that a generic implementation using SEAL network packets would look very inefficient compared to customized implementations that provide application-specific data compression [Cardell-Oliver 2013] or message suppression based on statistical techniques [Raza 2012]. At the moment we do not see an easy way out of this tradeoff between generality and efficiency of communications.

To the energy-conscious developer, SEAL and its compiler offers several features worth mentioning. First, there is the `turnOnOff` parameter that allows to power on hardware components only during their usage. This parameter is useful in case some infrequently-read sensors have high current consumption in the default, idle, mode. Second, there is the option to synchronize readings of multiple sensors in a way that minimizes the time the MCU has to spend in active mode (Section 3.3.3.4). There is also a caching mechanism (Section 3.3.3.3) that helps to avoid unnecessary extra reads.

3.5.3 Applicability evaluation

SEAL and its runtime are tuned specifically for sense-and-send type of applications. However, is can be used for other application archetypes a well (though it may not be the best choice for all of them!). The versatility of SEAL is demonstrated by the fact that it supports both periodic and event driven sampling and data transmission. It allows to write applications that include support for actuation, interactivity (queries can be defined and handled), data interpretation (using predefined functions and their combinations), and data aggregation across nodes.

3.5.3.1 Application classes

In order to evaluate the WSN application design space covered by SEAL, first we discuss whether and how SEAL supports the seven WSN application characteristics defined in [Bai 2009]:

- **Sampling** – both periodic (using `period` parameter) and event-driven (automatically for interrupt-based sensors, using event-specific `when` conditions for the rest of sensors).
- **Data transmission** – similarly, both periodic and event driven.
- **Actuation support** – yes. First, actuators can be activated or deactivated (with specific parameters), second, a system input can be linked to an actuator input (for example, analog-in port to analog-out port).

- **Interactivity support** – yes. Queries can be both issued and handled using SEAL. The former by using `output` statement with explicit list of fields to include and their values; the latter: by matching the query against a `when` condition.
- **Data interpretation support** – yes. SEAL features limited in-language support for new algorithm definition by combining pre-defined primitive functions (currently the default implementation includes 36 such functions).
- **Data aggregation support** – yes. SEAL allows defining data aggregation across multiple nodes, although the language has limited support for complex custom algorithm description (no loops or recursion).
- **Heterogeneous network support** – yes. First, SEAL applications are architecture-independent in the sense that they can be compiled for multiple architectures. Not all components are supported on all hardware, of course. Second, architecture-specific decisions can be described in the language.
- **Mobility support** – library level only (for example, GPS can be added as one of the sensors).

As it can be seen, SEAL at least partially supports all six out of seven design characteristics at language level. It can be used to program applications belonging to any of WSN application archetypes (as defined in [Bai 2009]). Therefore, SEAL can be classified as a WSN *application* programming language with universal applicability.

3.5.3.2 Concrete scenarios

We have identified a number of scenarios (Table 3.8) where it can be used to program the application logic. The scenarios were selected by two methods. There are four scenarios that are based on WSN deployments presented at high-impact conferences. The deployments are well-known in the community [Werner-Allen 2006] [Selavo 2007] [Ceriotti 2011] [Kothari 2007]. All the deployments are fairly traditional sense-and-send applications, but each with a particular twist. The remaining nine scenarios are based on problems that have been met in WSN and cyber-physical system projects in Institute of Electronics and Computer Science (EDI) and University of Latvia. They all belong either to realized or projected deployments.

The deployments roughly belong to three groups: sensor networks for environmental monitoring, functionality of “classical” embedded system software, including signal processing, and sensor networks or embedded systems for vehicular applications. Several belong to more than one group.

A distinction must carefully be made between the statement that *SEAL* supports a scenario and that *our software framework* fully supports a scenario. The first means that the application logic can be implemented using SEAL; the second means that there is also full support for OS level logic according to the pragmatic requirements of each scenario. We claim the former – that SEAL fully supports each scenario, as it is described here. We do not claim the latter, as that would

Table 3.8: SEAL application scenarios

Scenario	**Feature demonstrated**
Measuring microclimate [Elsts 2012a]	Periodic sampling
Controlling a watering system [Elsts 2012a]	Distributed processing
Collecting light measurements [Selavo 2007]	Interactive query support
Detecting seismic activity [Werner-Allen 2006]	Event-driven processing
Light monitoring in tunnels [Ceriotti 2011]	Intra-node proc. (using C)
Building a backup controller [Zviedris 2010]	Imperative elements
Analog signal processing [Strazdins 2011]	Intra-node processing
Generating a signal [Mednis 2012c]	Sequential execution
Detecting potholes #1 [Mednis 2011]	Intra-node processing
Detecting potholes #2 [Mednis 2011]	Intra-node processing
Detecting vehicle movement [Mednis 2012a]	Optimized intra-node processing
Finding parking space [Kothari 2007]	Distributed processing

require full implementation of many elements beyond the scope of this work, for example, LiteTDMA [Selavo 2007] protocol.

In total, twelve SEAL usage examples are presented. They demonstrate both practical applications of the language, and different features of SEAL. The language is used in sensor network and in other embedded system context. In most cases (for example, Listing 3.37), the implementation in SEAL code is straightforward after the algorithm description is given. However, in case many sensors are present, the code size is large, because it grows linearly with the number of components used; this is already hinted by Listing 3.27. Future improvements should be expected in this area. Some cases do not allow to be fully specified in SEAL itself (Listing 3.27 again) or can only be implemented syntactically (Listing 3.31, which require microsecond timers), because the system-level support is not there yet. Therefore the listings not only demonstrate the strengths of SEAL, but also suggest future research directions.

Microclimate measurements.

Scenario (Listing 3.21, 3.22): our microclimate monitoring system in precision agriculture (SAD) [Elsts 2012a] reads microclimate measurements on motes deployed in an orchard, stores the measurements locally and forwards them to the network towards of the base station. According to the wishes of agriculture scientists (the users of the system), the data can be sampled with frequency as low as one sample in 10 minutes.

The first version of the code (Listing 3.21) is a straightforward translation of the requirements. However, taking in account the experience from 2011 deployment and additional user wishes, we made a number of modifications (Listing 3.22).

The first line in Listing 3.22 tells that the system is to be used for extended periods of time. The peculiar requirements of SAD application come from the facts that, simultaneously, the application should be long-lived (we aim a lifetime that is equal to whole vegetation season), and the hardware has no support for real-time clock. Hence, the internal system uptime counter is used. The counter registers milliseconds since system's start and by default is 32 bits wide, which means that it wraps back to zero every 40 days. If `USE_LONG_LIFETIME` option is used, the counter becomes 64 bits wide. The code is slightly less efficient, but allows generating more useful timestamps.

The second line defines a *helper* actuator (LED in this case) that is used in line 9. Using this code helps domain experts to effectively discover failed nodes. Once again, the specific of SAD application is that the harsh conditions of the orchard sometimes cause sensor failures. In particular, the application sometimes freezes when trying to read SHT75 humidity sensor. Using the helper LED allows to visually diagnose such situations, as the LED is turned on before sensor reading is started, and turned off after it is successfully completed. If the LED remains bright, reading the sensors has failed.

Lines 4–8 defines a virtual light sensor. Not all motes in the deployment have the same sensors; most have the default light sensor for the SADmote hardware platform, but the one with address (ID) `0x0796` has more a precise SQ100 solar radiation sensor.

Line 9 tells that all the sensors should be read simultaneously, not using a different timer for each; this helps to avoid relative drifting of measurement intervals.

Finally, line 10 puts it all together, as well as tells that the sensors should be turned on before reading and off afterwards (`turnOnOff` parameter). In this way, additional energy savings are made.

The difference between the two versions illustrates the importance of real-world evaluation of high-level WSN programming approaches. Were it not evaluated in the field, the first version would look just fine; but in reality, it can be greatly improved (the second version).

Listing 3.21: Conceptual version of SAD application

```
1 // define system sensors
2 read Light, period 10min;
3 read Humidity, period 10min;
4 read Temperature, period 10min;
5 // define system outputs
6 output SdCard;
7 output Network, protocol CSMA;
```

Listing 3.22: Real-world version of SAD application

```
 1 config "USE_LONG_LIFETIME=y";
 2 define HelperLed RedLed, blink;
 3 // define system sensors
 4 when Variables.localAddress == 0x0796:
 5     define TheLight SQ100Light;
 6 else:
 7     define TheLight Light;
 8 end
 9 define AllSensors sync(TheLight, Humidity, Temperature);
10 read AllSensors, period 10min, associate HelperLed, turnOnOff;
11 // define system outputs
12 output SdCard;
13 output Network, protocol CSMA;
```

Controlling a watering system.

Among our future plans for SAD is to control a watering system automatically. The idea is that when the soil in an orchard or a greenhouse becomes too dry, a number of actuators should automatically turn on the water flow. Each actuator device receives data from multiple sensors in its single-hop neighborhood, and uses the minimal of soil moisture readings received to determine the course of action.

Listing 3.23 shows the straightforward code on a sensor device. The code on actuator devices (Listing 3.24) demonstrates how distributed processing can be implemented using SEAL. The values of `RemoteSoilHumidity` are taken from network; they correspond to humidity sensor readings on remote nodes.

Listing 3.23: On sensor device

```
// read soil humidity sensor
use SoilHumidity;
// send the measurements to radio
output Radio;
```

Listing 3.24: On actuator device

```
const MIN_HUMIDITY_THRESHOLD 20%;

// define the minimal humidity as minimum of values
// in last 60 seconds, max 20 value array
define MinHumidity min(takeRecent(RemoteSoilHumidity, 60s, 20));

when MinHumidity < MIN_HUMIDITY_THRESHOLD:
    // use watering system in this case
    use Watering;
end;
```

Collecting light measurements.

Scenario: light monitoring under shrub thicket in a remote island is performed. In order to make data retention and data delivery rates higher, data storage nodes are deployed throughout the network in addition to the regular sensor nodes. The regular sensor nodes produce data (light sensor measurements together with originator's address and timestamp) and send it out to the network. The storage nodes (Listing 3.25) overhear all radio traffic and log the data to SD card. LiteTDMA protocol is employed for efficient communication (line 11).

The example shows SEAL code on data storage node, as the sensor node code is trivial. The data on storage nodes is used only in case a data packet has not reached the base station. In this case, the base station issues a query asking for the lost data. The query includes address of the node whose data was lost and a timestamp that corresponds to the start of the missing data. The storage nodes process the query and respond to it (line 11) if the data is present (lines 13–14).

Listing 3.25: On data storage node

```
1 // On data storage node
2 const DATA_QUERY_COMMAND 1;
3
4 // remote light packet
```

```
read RemoteLight; read RemoteAddress; read RemoteTimestamp;
output File (RemoteLight, RemoteAddress, RemoteTimestamp), filename "LightData.bin";

// storage query packet
read RemoteCommand;
when RemoteCommand = DATA_QUERY_COMMAND:
    output Network, protocol LiteTDMA, file "LightData.bin",
        where
            Address = RemoteAddress
            and Timestamp >= RemoteTimestamp;
end
```

Detecting seismic activity.

Scenario: a sensor network is deployed on an active volcano. The network performs high-rate (100 Hz, multiple channel) seismoacoustic monitoring and detects events of interest (e.g. small earthquakes). Since the data rate is so high, it is not feasible to transmit all of the data to the base station. Triggered data collection is used instead, and data is downloaded from each sensor device following a significant earthquake or eruption.

Listing 3.26: On sensor node

```
 1 const COMMAND_EVENT_DETECTED 1;
 2 const COMMAND_REQUEST_DATA 2;
 3 const DATA_COLLECTION_INTERVAL 60s;
 4 const EWMA_COEFF_1 0.15;
 5 const EWMA_COEFF_2 0.2;
 6 const EVENT_DETECTION_THRESHOLD 0x1234;
 7
 8 // read sensors
 9 read AcousticSensor, period 10ms;
10 read SeismicSensor, period 10ms;
11 output File (AcousticSensor, SeismicSensor, Timestamp),
12     name "SensorData.bin";
13
14 // detect events
15 define CombinedSensor sum(AcousticSensor, SeismicSensor);
16 define EventDetectionFunction difference(
17     EWMA(CombinedSensor, EWMA_COEFF_1),
18     EWMA(CombinedSensor, EWMA_COEFF_2));
19 read EventDetectionFunction;
20 when EventDetectionFunction > EVENT_DETECTION_THRESHOLD:
21     output Network (Command), command COMMAND_EVENT_DETECTED;
22 end;
23
24 // respond to queries
25 when RemoteCommand == COMMAND_REQUEST_DATA:
26     output Network, file "SensorData.bin",
27         where RemoteTimestamp >= Timestamp
28         and RemoteTimestamp <= add(Timestamp, DATA_COLLECTION_INTERVAL);
29 end
```

The application logic (Listing 3.26) consists of three parts. First, the application samples acoustic and seismic sensors and stores the data locally (lines 9–12). Second, event detection algorithm is continuously run, as described in [Werner-Allen 2006] (lines 15–22). Node detects an event when the ratio between two exponentially weighted moving average (EWMA) functions becomes large enough. In this case the node sends event notification to the base station. If high enough number of nodes has detected an event at the same time, the base station issues data collection query. Third, the node handles data collection query command (lines 25–29). The command includes a timestamp. Data from the interval [timestamp; timestamp + 60 seconds] is sent back to the base station.

Light monitoring in tunnels.

Scenario: a wireless sensor and actuator network is deployed in a traffic tunnel. The sensor devices (Listing 3.27) sample light values, aggregate them, calculate average in 5 second period, and send it to network, where they are intercepted by tunnel control infrastructure, which in turn controls the artificial light level. Sampling frequency is location-dependent; interior nodes are sampled less often than motes in entrance and exit zones. SEAL uses a OS-level variable `moteIsInInterior` to determine whether it is the case (lines 13–17), assuming it can be set by management protocol or at compile time.

Listing 3.27: On sensor device

```
 1 const SAMPLING_INTERVAL 5s;
 2 const INTERIOR_REPORT_INTERVAL 5min;
 3 const ENTRANCE_REPORT_INTERVAL 30s;
 4 const SATURATED_VALUE 0xffff;
 5
 6 // Load C-based extensions for this scenario
 7 load "TunnelDataCollector.py"; // component description
 8 load "TunnelDataCollector.c";  // component implementatio
 9
10 // choose the sampling interval depending on system variables
11 // (assume that moteIsInInterior is a variable
12 //  that can be changed via management protocol)
13 when Variables.moteIsInInterior:
14     set reportInterval INTERIOR_REPORT_INTERVAL;
15 else:
16     set reportInterval ENTRANCE_REPORT_INTERVAL;
17 end
18
19 // there are 4 light sensors; ignore the ones that are saturated
20 define Light1 filterLess(LightWithId, SATURATED_VALUE), id 1;
21 define Light2 filterLess(LightWithId, SATURATED_VALUE), id 2;
22 define Light3 filterLess(LightWithId, SATURATED_VALUE), id 3;
23 define Light4 filterLess(LightWithId, SATURATED_VALUE), id 4;
24 // one measurement is an average of all valid light sensors
25 define LightAvg average(tuple(Light1, Light2, Light3, Light4));
26 // read the average value with 5 second period
27 read LightAvg, period SAMPLING_INTERVAL;
28 // use a custom data collector component
29 output TunnelDataCollector, interval reportInterval;
```

The application logic is the following [Ceriotti 2011]: "when reporting to the sink, the average s_{all} of all [non-saturated] values s_i is computed. For each s_i, if the difference $|s_{all} - s_i|$ differs from s_{all} by more than 50%, s_i is discarded and s_{all} recomputed." In our code, s_i corresponds to $\texttt{Light}_i$ (lines 20–23) and s_{all} (before recomputation) to `LightAvg` (line 27).

This example is interesting because it demonstrates the limitations of SEAL and how they can be overcome. The *re*calculation of s_{all} is hard to implement in SEAL due to scarcity of imperative programming constructs. We address this problem by defining application-specific extension: `TunnelDataCollector` component described in `TunnelDataCollector.py` file (line 7) together with its implementation in `TunnelDataCollector.c` file (line 8), which recalculates the value of s_{all}, if necessary, and sends it to actuator nodes.

The complete component library "glue" code is given in Listing 3.28 and consists of only 6 lines of Python code.

Listing 3.28: Component library code

```python
from testarch import *

class TunnelDataCollector(SealOutput):
    def __init__(self):
        super(TunnelDataCollector, self).__init__("TunnelDataCollector")
        self.useFunction.value = "tunnelDataCollectorSend(&tunnelDataCollectorPacket,␣sizeof(tunnelDataCollectorPacket))"

tunnelDataColl = TunnelDataCollector()
```

Building a backup controller.

EDI is developing a wild animal monitoring system [Zviedris 2010] (see also Section 2.7.1). The system consists of collars with embedded sensor hardware, supporting base stations placed in the forest, and a mobile data acquisition station. For reliability reasons, the collar devices are equipped with a second (backup) microcontroller. Listing 3.29 presents the code of this microcontroller (MCU).

The backup system has two inputs and two outputs. One input of the system is a pin that signals when the battery of the main MCU goes low, and another is a pin that is periodically toggled to signal that the main system is still alive. The outputs of the system are a pin that allows putting the main MCU in reset, and a pin that allow to completely disabling it.

The logic, in outline, is to detect situations when the main MCU should be reset or turned off, and act correspondingly. In addition, the backup system is equipped with a radio that transmits the carrier signal periodically (`CarrierSignalGenerator` component in code). The frequency of the carrier signal transmissions allows detecting whether the main MCU is alive or turned off.

Listing 3.29: On backup device

```
const RESET_PORT 1; const RESET_PIN 1;
const POWER_PORT 1; const POWER_PIN 2;
const BATTERY_LOW_PORT 1; const BATTERY_LOW_PIN 3;
const WATCHDOG_PORT 1; const WATCHDOG_PIN 4;

define ResetOut DigitalOut, port RESET_PORT, pin RESET_PIN;
define PowerOut DigitalOut, port POWER_PORT, pin POWER_PIN;
define BatteryLow DigitalIn, port BATTERY_LOW_PORT, pin BATTERY_LOW_PIN;
define Watchdog DigitalIn, port WATCHDOG_PORT, pin WATCHDOG_PIN, interrupt, risingEdge;

// The logic is simple:
// when battery of the main controller is low, turn it off,
// but turn it back on after a while (however, limit the number of tries)
read BatteryLow;
when BatteryLow:
    use PowerOut, off;
else:
    do, initialTimeout 10min, once:  // add 10 minute hysteresis
        use PowerOut, on;
    end
end

// check for watchdog signal, and if none received, reset the main MCU
const WATCHDOG_MAX_TIME 20s; // seconds
const MAX_RETRY 5;
when not Watchdog:
    // at the end of 10 seconds: turn power off
    do, period WATCHDOG_MAX_TIME, times MAX_RETRY:
        use ResetOut;
    then:
        // give up
    end
end
```

Analog signal processing.

EDI was a participant in the first Grand Cooperative Driving Challenge [Strazdins 2011]. The vehicle we used was a modified Mazda 6, which had an option to electronically control accelerator pedal. By default the pedal receives input directly from car's electronic control block. Our modification was to include a Tmote Sky sensor device in the path. The mote functions as a bridge (Listing 3.30); it has wires attached to its ADC and DAC ports; as resistors are used, the input signal is range is different from the output signal range and has to be compensated in the program. In addition, the output wire must supply at least minimum if 1.6 V voltage. In addition, the application logs the signal values; they can be used to analyze and replay the usage of the accelerator.

Listing 3.30: On controller device

```
// define input signal channel
define InputPin AnalogIn, channel 6;
// define output signal channel
define OutputPin AnalogOut, pin 3, port 3;
// define input->output mapping: input range is 120..255, output: 0..100
define MySignal map(InputPin, 120, 255, 0, 100), out OutputPin;
// define the action
use MySignal;
```

Generating a signal.

Scenario: in order to understand a communication protocol, an electrical engineer has to regenerate a test signal pattern that was recorded in an experimental measurement. The signal is binary (on/off), has microsecond gradation.

In order to describe this signal in SEAL (Listing 3.31), a `pattern` construct is used. The signal is repeated periodically every 100 milliseconds. In order to implement this feature, two nested `do` statements are used. The outer statement is a wrapper that contains all the logic; it specified with no parameters, therefore is executed continuously and periodically. The logic of the inner statement is executed just once per every execution of the outer statement. Before the execution of the inner statement there is always 100 millisecond timeout. The inner statement could also be written in another way, without the `initialTimeout` parameter; in that case `then` branch would have to be added, with empty logic and 100 millisecond duration.

Listing 3.31: On generator device

```
// define a signal using pattern declaration with microsecond intervals
pattern P (300us, 100us, 300us, 400us, 300us, 200us);
do:
   do, once, initialTimeout 100ms:
      // output this pattern to a specific pin
      use DigitalOut, port 1, pin 6, pattern P;
   end
end
```

Detecting potholes #1.

Scenario: a vehicle is equipped with a sensing device that records accelerometer readings continuously. The purpose of the sensing device is to detect potholes (irregularities in road surface). Multiple detection algorithms are possible [Mednis 2011] (see also Section 2.7.2).

The idea of STDEV algorithm (Listing 3.32) is to look at standard deviation of the latest readings of accelerometer's Z axis, and declare a pothole detected when STDEV is larger than the threshold.

Z-DIFF algorithm (Listing 3.33) applies thresholding to the difference of last two Z axis measurements. It is implemented as a special case of STDEV algorithm, using the mathematical relation between standard deviation of the last two measured values and the difference between them.

Threshold algorithm (Listing 3.34) simply detects a pothole when absolute values of Z axis exceeds or falls below some specific thresholds.

Finally, G-ZERO algorithm (Listing 3.35) is based on the observation that driving into a pothole leads to a short-time free fall effect. As known, during free fall there is no acceleration. Regardless of the accelerometer position, measurements on all three axis are going to be close to zero.

In all algorithms the accelerometer is read with 10 Hz frequency.

Listing 3.32: STDEV algorithm

```
const ACCEL_Z 2; // channel number for Z axis

const THRESHOLD 100;

define AccelZ AnalogIn, channel ACCEL_Z;
define Deviation stdev(take(AccelZ, 10));

read Deviation, period 100ms;
when Deviation > THRESHOLD:
   use RedLed, on;
   use Beeper, on, duration 200, frequency 1000;
else:
   use RedLed, off;
end
```

Listing 3.33: Z-DIFF algorithm

```
const ACCEL_Z 2; // channel number for Z axis

const THRESHOLD 100;

define AccelZ AnalogIn, channel ACCEL_Z;
define Difference multiply(stdev(take(AccelZ, 2)), 2)

read Difference, period 100ms;
when Difference > THRESHOLD:
   use redLed, on;
   use Beeper, on, duration 200, frequency 1000;
else:
   use redLed, off;
end
```

Listing 3.34: Threshold algorithm

```
// Z axis measurements:
// 1680 - norm        (1g)
// 2080 - weightless (0g)
// 1280 (assumed) -  (2g)
const THRESHOLD_LOW 1360;
const THRESHOLD_HIGH 2000;

// channel number for Z axis
const ACCEL_Z 2;

define AccelZ AnalogIn, channel ACCEL_Z;

read AccelZ, period 100ms;
when AccelZ < THRESHOLD_LOW or AccelZ > THRESHOLD_HIGH:
   use RedLed, on;
   use Beeper, on, duration 200, frequency 1000;
else:
   use RedLed, off;
end
```

Listing 3.35: G-ZERO algorithm

```
// 0g is ~2080 accel value.
const THRESHOLD_LOW 1930;
const THRESHOLD_HIGH 2230;

// channels for X, Y, and Z axis
define AccelX AnalogIn, channel 0;
define AccelY AnalogIn, channel 1;
define AccelZ AnalogIn, channel 2;

read AccelX, period 100ms;
read AccelY, period 100ms;
read AccelZ, period 100ms;
when AccelX > THRESHOLD_LOW and AccelX < THRESHOLD_HIGH
        and AccelY > THRESHOLD_LOW and AccelY < THRESHOLD_HIGH
        and AccelZ > THRESHOLD_LOW and AccelZ < THRESHOLD_HIGH:
   use RedLed, on;
   use Beeper, on, duration 200, frequency 1000;
```

```
else:
    use RedLed, off;
end
```

Detecting potholes #2.

Pothole detection described in the previous scenario can be made more precise; the noise can filtered out before running the detection algorithms. Accelerometer is typically read with relatively high frequency (100 Hz reading is typical) on a budget commercial-off-the-shelf hardware, therefore high frequency noise is present, and low-pass filtering can help. In this example scenario (Listing 3.36), data is output to serial port for later off-line processing instead of applying an online detection algorithm as in previous example.

Listing 3.36: On intra-vehicle sensor device

```
// constants
const NUM_SAMPLES 5;
const ADJUSTMENT_WEIGHT 5;
const ACCEL_CHANNEL 2;
// define our accelerometer sensor
define Accelerometer AnalogIn, channel ACCEL_CHANNEL;
// define accelerometer sensor with low-pass filter applied
define SmoothedAccelerometer smoothen(Accelerometer, NUM_SAMPLES,
    ADJUSTMENT_WEIGHT);
// read the noise-removed sensor
read SmoothedAccelerometer, period 10ms;
// output to serial port
output Serial, aggregate;
```

Detecting the state of a vehicle.

Scenario: the device from previous scenarios is used to detect whether the vehicle is in driving or standing mode. Such a detection has practical uses, for example, for car companies. The algorithm (Listing 3.37) cannot expect complete absence of movement even from a vehicle, as small movements are possible due to engine vibrations and other factors even when the vehicle is standing. Furthermore, some hysteresis should be present to “lock” a vehicle in its current state and avoid “jumping” between the states. The algorithm is as follows [Mednis 2012a]:

1. sample accelerometer readings on all three axis with 100Hz frequency for a second;
2. calculate standard deviation of the readings on each axis;
3. sum the deviations and compare them to predefined set of thresholds (must be calibrated or determined otherwise for each vehicle):
    - if the sum is more that $threshold_{high}$, set the detected mode to “moving”.
    - else if the sum is less than $threshold_{low}$, set the detected mode to “standing”.
    - else the detected mode is not changed.

Listing 3.37: On intra-car sensor device

```
const THRESHOLD_LOW  50;
const THRESHOLD_HIGH 100;

set isMoving False; // vehicle mode

define AccelX AnalogIn, channel 1, period 10ms;
define AccelY AnalogIn, channel 2, period 10ms;
define AccelZ AnalogIn, channel 3, period 10ms;

define X_dev stdev(take(AccelX, 100));
define Y_dev stdev(take(AccelY, 100));
define Z_dev stdev(take(AccelZ, 100));

define SDev sum(X_dev, Y_dev, Z_dev), lazy;

read SDev;
when SDev > THRESHOLD_HIGH:
    set isMoving True;   // movement detected
elsewhen SDev < THRESHOLD_LOW:
    set isMoving False;  // absence of movement detected
end
```

Finding parking space.

The goal of this application is "to identify a sensor node with a free spot that is as close to the desired destination of the driver as possible" [Kothari 2007]. Since there is no macroprogramming support in SEAL, the application is divided in two separate parts: the part that is executed on stationary nodes (corresponding to parking spaces) and on mobile nodes (corresponding to would-be parked vehicles). The code is shown in Listing 3.38 and Listing 3.39, respectively.

The free-space search algorithm uses ever-expanding search space (limited by the maximum number of hops a routing protocol supports). The example relies on system-level support for multihop broadcast and unicast communications.

The reservation is done in two steps. In the first, the parking node receives a query message, and, in case it is free, responds back with a reply message. A temporary reservation (for one second) is made. If during this time reservation confirmation message is received, the node advances to "permanently reserved" state. Otherwise it falls back to free state.

The mobile node queries its neighborhood for free parking locations. If no such location is found, the search space is enlarged and a new iteration starts.

The main drawback of this SEAL implementation is that compared to the Pleiades version [Kothari 2007], this algorithm is not as resilient to communication timeouts. However, unlike the TinyOS versions, it is both reliable and efficient. The first property is entailed by the fact that if communication succeeds, one and only one reservation will always be made when there is free space available. The second property by the fact that the search space is incrementally expanding.

Listing 3.38: On stationary ("parking space") nodes

```
const CMD_QUERY_STATUS 1;
const CMD_REPLY_STATUS 2;
const CMD_RESERVE 3;
```

```

set IsFree true;     // is free for new reservations?
set IsWaiting false; // is waiting for reservation confirmation?

when RemoteCommand = CMD_QUERY_STATUS and IsFree:
    // move to "waiting" state
    set IsFree false;
    set IsWaiting true;
    output Network (Command, Status), command CMD_REPLY_STATUS, Status
        IsFree;
end

when IsWaiting:
   do, initialTimeout 1s:
       // fall back to "free" state
       set IsWaiting false;
       set IsWaiting true;
   end
end

when RemoteCommand = CMD_RESERVE:
    // move to "reserved" state
    set IsWaiting false;
end
```

Listing 3.39: On mobile ("vehicle") nodes

```
const CMD_QUERY_STATUS 1;
const CMD_REPLY_STATUS 2;
const CMD_RESERVE 3;

set HasReservation false; // whether a place to stay is found
set Hops 1;               // how far the queries will go

read HasReservation;
when not HasReservation:
    output Network (Command), command CMD_QUERY_STATUS, hopcount Hops;
    set Hops add(Hops, 1);
end

when RemoteCommand = CMD_REPLY_STATUS and not HasReservation:
    output Network (Command), command CMD_RESERVE, destination
        RemoteAddress;
    set Hops 1;
    set HasReservation True;
end
```

3.5.4 Empirical usability evaluation

In order to evaluate the usability of our software we conducted two preliminary user studies.

We were looking for answers to the following questions:

- Can *novice programmers* successfully develop applications in SEAL?
- Can *novice WSN users* successfully develop applications in SEAL?
- Can programmers understand the intent of unfamiliar code written in SEAL?

- How does the developer success rate in SEAL compare with that of existing WSN languages?
- What are the specific difficulties in using SEAL?

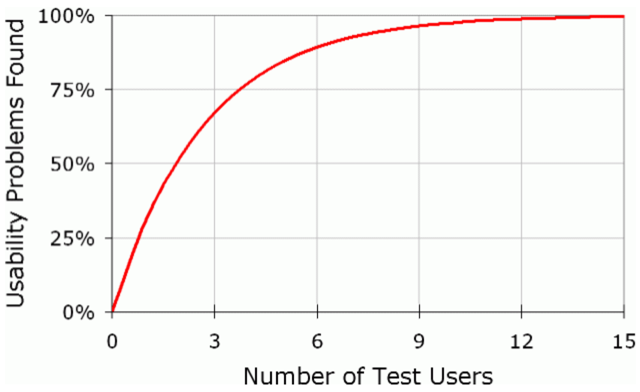


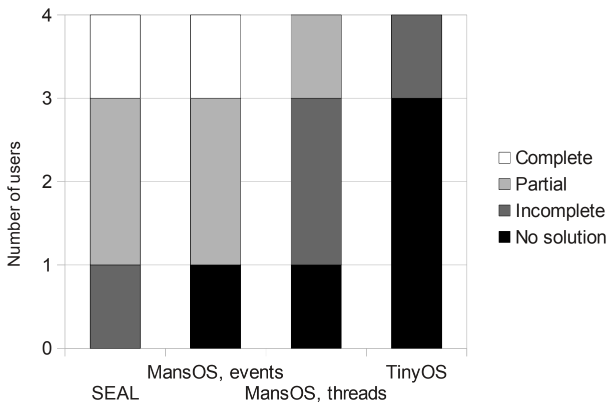
Figure 3.9: Usability problems found as a function of number of test users [Nielsen 2000]

What is the minimal sample size that allows to achieve good results? Before the end of the 20th century usability testing was treated as formal process and employed methods typically used in research experiments [Barnum 2010]. This allowed to apply statistical analysis on the results, but made the experiments expensive, time consuming, and requiring many participants. More recently, researchers determined that the law of diminishing return applies very strongly here: “the maximum cost-benefit ratio, derived by weighing the costs of testing and the benefits gained, is achieved when you test with three to five participants” [Barnum 2010]. The Fig. 3.9 shows that even a single user helps greatly, and that increasing the number of users above five leads to smaller improvements. However, the curve only applies to a *particular* usability test; the number of tests required to find all the usability problems in a product is very large. Therefore it is more profitable to conduct several tests with fewer users than a single test with a lot of users. Our first study included 8 users (6 valid results); our second study included 14 users (12 valid results) in 4 groups (3 valid results in each group).

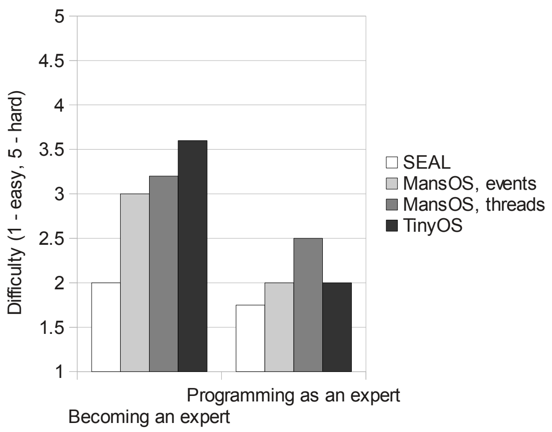
The first study took place in November 2011. We used the MansOS usability evaluation methodology (Section 2.6.6) as the basis for this study as well.

Firstly, four additional third year computer students (with no experience in WSN) were asked to implement the exercise in SEAL. The results are shown in Fig. 3.10 (*cf.* with Fig. 2.27 and discussion in Section 2.6.6). For first-time user with background in computer science, the usability of SEAL is better than that of TinyOS (we calculated $P < 0.02$). SEAL also scored higher than MansOS, although the difference in this four-user test was not statistically significant. The users also expect that it is going to be much easier to become an “expert” SEAL programmer than “expert” MansOS or TinyOS programmer.

Secondly, four agriculture scientists from Latvia State Institute of Fruit-Growing were asked to program the same solution using SEAL GUI. After initial help with



(a) Student solutions



(b) Difficulty estimation

Figure 3.10: The results of the first SEAL usability study

installation and setup, all were able to complete this assignment on their own or using only discussions with colleagues. (However, only 2 results were finally submitted to instructors e-mail.) Such as result would be unthinkable with C, where they would have to battle with issues such as C pointer semantics and memory alignment.

There were number of issues with SEAL syntax that created difficulties, leading to solutions that were partially correct – the programmer's intention was clear, but, for example, brackets were not used where they should have been. After the tests these syntax problems were ruled out, leading to an updated version of SEAL grammar.

The second study took place in October 2012. We adopted methodology and exercises from the SensorBASIC user study [Miller 2009].

For evaluation we recruited computer science undergraduate students with no WSN programming experience (the "intermediate programmers"), as well as students and scientific staff from other domains with little or no programming experience (the "novice programmers"). The participants were given access to computers with IDE installed (SEAL or TinyScript, depending on their group), and attached sensor devices equipped with light, temperature and humidity sensors.

At the start of the test, the participants were instructed to read the tutorial and study the IDE for 20 minutes. After this time, four exercises were revealed. Each participant was given 70 minutes to complete them. At the end they had to send in their solutions to instructors e-mail, as well as fill in and hand in a questionnaire documenting their specialty and previous programming experience.

For the novice programmers, we were primary interested in the users ability to complete the tasks at all. For the experienced programmers, we were also interested in success rate comparison with TinyScript.

The first three exercises asked to code specific applications (adopted from [Miller 2009]):

Table 3.9: Exercise completion success rate

	Ex. 1	Ex.2	Ex.3	Ex.4	Total, %
TinyScript, intermed.	3/3	3/3	3/3	3/3	100 %
SEAL, intermed.	3/3	3/3	3/3	3/3	100 %
SEAL, novice 1	3/3	0/3	2/3	n/a	55.6 %
SEAL, novice 2	3/3	2/3	2/3	2/3	75 %

1. Blink a LED with 2 second period (Listing 3.40);
2. Send a message to the base station if the light sensor is covered (Listing 3.41);
3. Turn on the LED if and only if the light sensor is covered (Listing 3.42).

The fourth exercise asked to "describe the meaning of the code in Listing 3.20".

Listing 3.40: Solution to exercise 1

```
use RedLed, period 2s;
```

Listing 3.41: Solution to exercise 2

```
when Light < 100:
   use Print, format "No␣light", out Radio;
end
```

Listing 3.42: Solution to exercise 3

```
when Light < 100:
    use RedLed, on;
else:
    use RedLed, off;
end
```

From the 14 participants we recruited, two were unable to test their solutions to programming exercises because of hardware or system problems. That left us with twelve: six computer science students, three physicists, two agricultural scientists, and one electrical engineer; seven were undergraduate students, five – graduate students or scientific staff. The results of the study are given in Table 3.9.

The first exercise was completed by all subjects.

Second and third exercises were completed by all computer science students in both groups. However, the second was not completed by anyone in "novice 1" group. The translation of the exercise was misguided and suggested to do something that in fact was impossible in SEAL at that moment. After this group we reformulated the exercise, improved SEAL documentation, added printing to network (used in Listing 3.41), and the results were improved.

The third exercise was completed by most, although some SEAL users had included unnecessary `read Light` statement before `when` statement. Seeing these results, we decided to modify the semantics of the language and require explicit reading even for sensors that are used only in conditions. (This feature is included in the current, May 2013 version of SEAL.)

The fourth exercise was completed by all who attempted it.

The study demonstrates that the computer science students with no WSN experience succeeded in all (12 out of 12) cases when using SEAL, and the novice programmers succeeded in $\frac{2}{3}$ (14 out of 21) cases using SEAL.

Our results are comparable with outcomes from [Miller 2009] (54 %, 45 %, and 46 % in the group with no programming experience, 100 %, 89 % and 67 % with limited programming experience). The user study from [Bai 2009] (three different exercises) similarly reported 40.6 % average success rate (although WASP and WASP2 had 80.6 % average).

We draw attention to the fact that solutions in SEAL are ~35 % shorter than in SensorBASIC (9 versus 14 lines in total) and contain fewer syntactic elements.

Most of the results form the questionnaire show no statistically significant differences. The exception is SEAL IDE, which was perceived as more easily usable than TinyScript IDE ($P < 0.05$). The average mark was 5 out of 7 for SEAL IDE and 2 out of 7 for TinyScript IDE.

Research suggestion that “control structures are hard for novices to grasp” [Pane 1996] was confirmed: many novice users struggled with syntax of `when` statements.

Archetype-specific languages as WASP are smaller than SEAL, therefore they can be faster to learn and more successful in tests like this. Nevertheless, [Bai 2009] suggests the results would be improved if we provided either code templates (as in WASP or TinyTemplate), or forced visual-only programming (as in WASP2 or SwissQM).

The IDE has a menu item for browsing example applications. We did not encourage the participants to use it; we feared that it would make the task too trivial. In retrospect, that was a mistake. None from the first study group had discovered this menu item for themselves, and were excited about this feature when informed about it after the study. Many users also suggested using more examples in the documentation.

The users described the visual edit window as helpful and frequently used it during the study.

3.6 Concluding remarks

This chapter addresses the problems faced by novice programmers as sensor network application developers. Using lessons from novice programming theory, we show that it is possible to design an easy-to-learn WSN application description language: SEAL, and develop its compiler and component library. The language and its compiler includes a number of features not present or not common in the current WSN node-level programming languages, for example, custom suffixes for mapping between physical units and raw sensor measurements, easily extensible component library (including application-specific extensions), and fully transparent low power mode handling. SEAL offers the *virtual sensor* abstraction instead of the function abstraction, as the former has more direct real-world counterparts (physical sensors).

SEAL allows to write compact and simple code, is compiled to efficient binary code (less than 40 % resource usage overhead), and is suitable for typical WSN usage scenarios. The syntax of SEAL is shown to be simpler and the grammar smaller than those of general purpose programming languages, such as C. In order to increase the usability of SEAL and compare it with similar languages, we conducted multiple user studies. The studies both lead to improvements in the language, and showed that the existing version of the language already demonstrates good usability.

Applying formal methods (such as model checking) to analyze the source code of SEAL in order to increase the reliability of its applications is one direction of future work. A sensor network specific language like SEAL is much more amenable to compile-time analysis and correctness deduction, compared with C. Its compiler is smaller (therefore the number of its semantic rules also is smaller), and more accessible for modifications. For example, the result of such a formal analysis of a particular application could be a comprehensive test set of this application that would allow to trace all possible paths of execution.

It is likely that there are many sensor network users who do not want to learn programming even in its simplest forms, but instead prefer to use web-based forms to configure their sensor network applications. In order to take into account their needs, hypertext form generator should be added to the SEAL compiler. It would generate application-specific hypertext form with input fields that would allow to configure variables of interest, such as sensor read intervals.

Another item of future work: better support for transformations between different notations (metamodels). For example, at the moment, the transformation between SEAL-Blockly blocks and SEAL code is possible only in one direction (from visual to textual representation); support for reverse transformations should be added. There is no need to stop here – other notations and presentations may be added, since no notation is universally preferable.

Last, but not least item of future work: macroprogramming support. Macroprogramming is an important area of WSN programming, and SEAL support for it is limited. Firstly, at the moment there is no option to *pull* the data from remote nodes, therefore client-server logic must be used to implement distributed algorithms in SEAL. Secondly, as mentioned in Section 3.5.1, the compiler can generate code for multiple roles in the network, such as “base station”, “sensor node”, and “data forwarder”; however, using a full-scale macroprogramming framework together with SEAL would provide the user additional abstractions and mechanisms, such as code migration from node to node, adaptive self-optimizations, logical neighborhoods. It would also allow to leverage existing implementations of complex algorithms: serialized access to variables on neighboring nodes, distributed locking and distributed resource arbitration. This is another research direction.

Chapter 4

A sensor network use case

Contents

4.1 Introduction

Accurate and easily accessible microclimate data coming from spatially distributed range of locations is useful for both for agricultural scientists researching the effects of various environmental parameters in an orchard, and for agriculture practitioners willing to increase yield quantity and quality from their plantations.

SAD is a wireless sensor network application for collecting microclimate data in precision agriculture. This chapter describes the custom hardware solution, the software created and adapted, and the three pilot sensor network deployments carried out in the orchard of Latvia State Institute of Fruit-Growing. The first pilot sensor network was deployed in 2009, used 8 motes, and harvested data for a month; the second deployment took place in 2011, used 17 motes (12 of them custom-made) and lasted for one and a half month; the third deployment took place in 2012, used maximum of 19 motes (18 of them custom made), and lasted for six and half months, from April to November.

In SAD project our research group experimentally evaluates suitability of wireless sensor networks for microclimate (light, temperature, humidity) monitoring in precision agriculture. This task is carried out in collaboration with Latvia State Institute of Fruit-Growing (LSIFG), as part of a greater precision agriculture research project "Development and adaptation of innovative risk reducing technologies for fruit and berry growing in conditions of Latvia". Informal name of the project is "Sensors in Fruit Garden", which has the Latvian abbreviation "SAD". The name has good suitability for our project, as in Russian "sad" means "orchard".

The work on SAD project was primarily done by a research group formed by the author, Rihards Balass and Janis Judvaitis, working under the guidance of Leo Selavo. This chapter uses materials from [Elsts 2012a] and [Elsts 2012b].

4.2 The motivation

Microclimate factors have tremendous influence on fruit crop yield quantity and quality. For instance, the fruits that have received smaller amount of sunlight during the growth process are going to have lower quality and market value, because of reduced sugar contents and less attractive coloring (for example, apples being sour and green, rather than sweet and red).

In past, these microclimate factors were largely unmanageable, but this has changed with the proliferation of protective covers for fruit species (such as apples, sweet cherries, strawberries and raspberries), precise watering systems, and other precision agriculture technologies. In the modern orchard, microclimate can be controlled and changed. For example, by using covers, the fruits are protected from rain and hail that can damage the yield. On the other hand, the cover itself can have a negative influence on the yield – it not only protects the plants and helps to establish optimal moisture conditions, but also reduces the amount of sunlight that plants receive, and is beneficial for development of pathogens and pests.

It is essential to have a high-quality microclimate monitoring solution in place to fully exploit the options offered by such a microclimate control system, register its own effects, and to provide near real-time alarm signals in case some of parameters reach values dangerous either for the yield or the plants themselves. Wireless sensor networks is a technology that naturally fits in this role, and as such have been frequently used in precision agriculture [Baggio 2005] [Wark 2007] and environmental monitoring [Tolle 2005] [Selavo 2007] applications.

Several distributed microclimate monitoring systems have been built and used in Latvia. One of the first, [Baums 2007] describes a 1-wire sensor based system used in the Botanical garden of University of Latvia. There is also the bee-hive monitoring system [Zacepins 2010] developed in Latvia University of Agriculture. However, literature survey did not show any other publication describing a *wireless* sensor networks deployed in Latvia for this purpose up to 2012, except ours. Using wireless communication allows to significantly expand the network, but makes the system construction much more difficult, especially if multihop routing has to be used. Recently this challenge has also been taken up by Institute of Mathematics and Computer Science of the University of Latvia, where a sensor network system for cranberry field monitoring is developed [Alberts 2013].

Back in 2009 the researchers at Latvia State Institute of Fruit-Growing (LSIFG) [LSI 2013] were looking for a way to record and store micro-climate data at various points of the institute's orchard. Before the start of SAD project, microclimate measurements from the orchard were collected manually using data-loggers. This method was labor intensive and offer limited precision, as well as made it impossible to achieve synchronous sensor reading in multiple locations. Our research group saw that it was possible to get rid of these limitations by deploying a sensor network.

The author and Girts Strazdins conducted a pilot research experiment in 2009 that used eight Tmote Sky sensor devices with built-in sensors. The sensors measured humidity, total solar radiation, and photosynthetically active radiation. The data was collected by a single-hop sensor network. This network was active from August 17th until September 15th, 2009. A total of 661 hours of real time data included 405 061 measurements collected at 20 second intervals.

From this initial experiment several lessons were learned. Firstly, the author become convinced that custom sensors are necessary for this application. The default light sensors located on Tmote Sky are not suitable for agricultural monitoring due to their limited dynamic range. During sunlight the sensors become saturated, yielding inaccurate data about the amount of radiation received by the plants. Furthermore, the spectral response of the photosynthetically active light sensor was very different from the actual spectral response of photosynthesis process in the plants.

Also, the agroscientists desired higher resolution than Tmote Sky could provide with the 12-bit ADC. Although the SHT11 sensor had sufficient precision, it was located on-board, inside the weatherproof box, where humidity and temperature values are significantly different from the conditions outside, in the orchard.

Secondly, a significant proportion of the pilot measurements were lost due to erratic radio links. Packet delivery failures, being frequently encountered in WSN, ought to be taken in account as one of design considerations. If the data was stored on the sensor nodes, using a flash storage chip or card, it could be collected manually at a later time.

Thirdly, a network with multiple hops is necessary. The ability to attach an external antenna and use radios with higher transmission power could be useful as well, at least for a second-tier, long-haul network.

4.3 Hardware

To face the challenges outlined by our pilot study one could purchase an existing sensor platform and extend it with appropriate sensors. This has the benefit of simplicity, however, this may also result in suboptimal extendibility, energy consumption and cost.

A TelosB-compatible [Polastre 2005] sensor device, such as Tmote Sky, is a popular first choice due to its ultra-low energy consumption, versatility, and reasonable 100$ price range. Tmote Sky has Texas Instruments MSP430F1611 MCU, IEEE 802.15.4 compatible radio transceiver, 1MB external flash memory chip, and 16 pin expansion port with ADC and digital inputs. However, the size of the flash may be insufficient for long-term micro-climate data. The storage capacity could be increased by attaching a larger data storage entity, such as an extension shield with SD card slot. Unfortunately, the MSP430 SPI interface is not fully exported, making it hard to interface with SD card. Another drawback is the 2.7 V power voltage required for successful flash data storage, limiting the useful battery lifetime.

Another option was to use a commercially available off-the-shelf (COTS) embedded device such as Arduino [Arduino Team 2012], with accompanying extension shields for sensors and communication. The cost would be reasonable. However, the target application would require many extension shields increasing the cost and complexity. Arduino has significant drawbacks in comparison with Tmote Sky – higher energy consumption and less resources (RAM, flash memory, IO ports). Therefore, Arduino is as platform more suited for hobbyist projects and rapid prototyping rather than for long-living sensor networks.

A reasonable alternative is a COTS sensor mote outfitted for agricultural monitoring such as Waspmote [was 2012]. Using such a device would reduce the amount of the required to deploy the network. However, the cost is much higher: 135 Euro for the mote and 250 Euro for Agriculture PRO extension board (the official prices in Libelium home page in 2011), not counting additional equipment, such as custom rechargeable batteries and enclosures. Also, Waspmote has higher energy consumption than MSP430-based devices. It has Atmega1281 MCU, which is one of the largest and most energy hungry MCUs in Atmega family. However, these drawbacks are partially countered by the expected simplicity and ease-to-use. Libelium provides simple API and software library for radio communication and for accessing various sensors. In the end, our research group purchased eight Waspmotes with corresponding extension boards and sensors, and five more Waspmotes without sensors.

The final option is to design a custom hardware. A significant number of WSN deployments still design their own hardware, citing for the following reasons: "maximum power efficiency" [Dyo 2010], "specific sensor, size and power requirements" [Suh 2006], "specific sensing needs" [Franceschinis 2009] and "specific packaging" [Detweiler 2010] [Dyo 2010]. In our case, by leaving out the components nonessential to our application (such as USB interface, LDO, on-board sensors), the cost and energy efficiency can be optimized. Also, by limiting the number of components the design becomes more robust.

The design of SADmote has gone through multiple revisions (Table 4.1), gradually including number of incremental improvements. The second version (SADmote v02) is the one our research group used for the deployment in 2011, and the third version (SADmote v03) is the one used for the deployment in 2012 and (with small modifications) in 2013.

Table 4.1: Comparison between Tmote Sky and different versions of SADmote

	Tmote Sky	**SADmote**		
		2011	**2012**	**2013**
Microcontroller	MSP430F1611	MSP430F1611		
Radio	CC240	MRF24J40	AMB8420	AMB8420
Radio freq. band	2.4 GHz	2.4 GHz	868 MHz	868 MHz
Ext. memory size	1 MB	2 MB	2 GB+	2 GB+
LED	Three	One	Three	Three
Onboard sensors	Light, Temp., Humidity	None		
Programming interfaces	USB, JTAG	SD-card, JTAG		
Real Time Clock	No	No	No	Yes

The SADmote (Fig. 4.1) is built on a two-layer printed circuit board and fea-

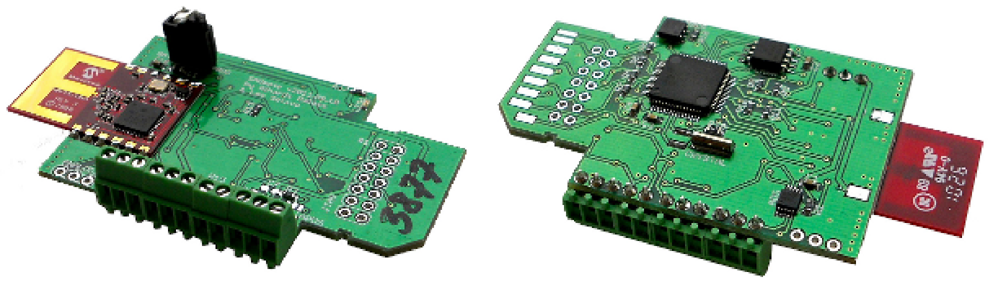


Figure 4.1: SADmote v02, front and back views

tures ultra-low energy consumption MCU MSP430F1611 (code memory size 48 kb, RAM size 8 kb). The MCU was selected as a true-and-tried design choice (also used on Tmote SKy), and because our software already had support for it. As for the radio, Microchip's MRF24J40MA radio module was initially selected (2.4–3.6 V operational voltage, 18 / 22 mA Tx/Rx current consumption, -94 dBm sensitivity). Even though the chip used in this module (MRF24J40) has slightly worse characteristics than, for example, CC2420 transceiver chip used by Tmote Sky (2.1–3.6 V, 18.8 / 17.4 mA, -95 dBm), it was a first choice. The reasons: first, our research group lacked the experience to build our own radio module; second, other available radio modules (e.g. Amber Wireless AMB2720) were approximately two times more expensive. MRF24J40MA also has the benefit that it can easily be replaced by other modules from the same manufacturer, such as MRF24J40MB-I, which features higher Tx power, or MRF24WB0MB, which has external antenna connector – these options would be useful for construction of a second-tier, longer haul network.

However, the Microchip radio, although behaving good in synthetic tests (Fig. 4.2, tested using nine receiver & one transmitter SADmote v02 motes in line-of-sight conditions and Fig. 4.3, tested using ten SADmote v02 motes in a hall), failed to demonstrate the required stability in real world conditions. The reasons included high radio signal absorption by tree foliage, and periodical freezing of the system, with probability proportional to number of packets received and transmitted interchangeably. One the reviewers of our SADmote article [Elsts 2012b] pointed at Amber Wireless radio modules; for new revisions of SADmode, AMB8420 was selected, because of this recommendation and the fact that modules with sub-GHz frequency were available.

The first versions of SADmote had AT25DF161 flash chip (2 MB storage size). The storage size was sufficient for our current needs (maximum 6 full month deployment, restricted by the length of the vegetation season in Latvia), but did not leave large margin of safety for longer deployments. More importantly, the chip was very cumbersome to use: due to severe serial interface speed limitations, data extraction from a single mote could take as long as 20 minutes. Therefore newer version of SADmotes feature 2 GB and 4 GB micro-SD cards. The card can be easily removed or replaced within seconds; afterward the data can be extracted by using any SD card reader connected to a PC.

SADmotes also have ADS111x 16-bit ADC chip (with one or two analog inputs)

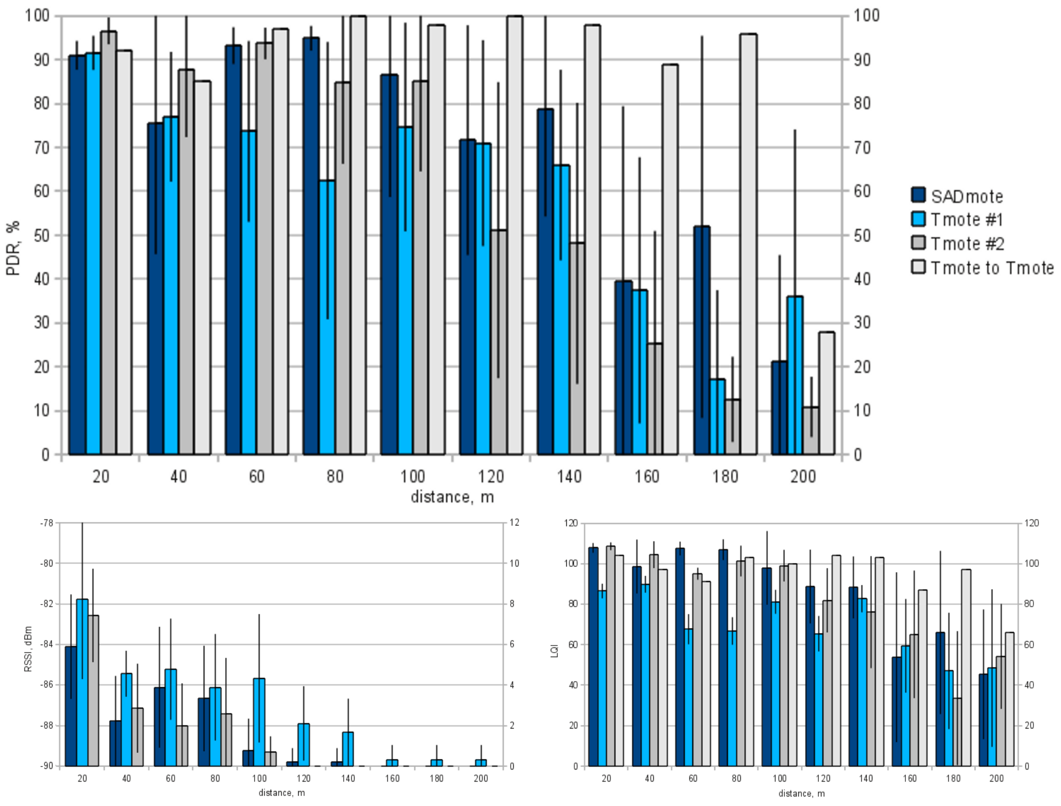
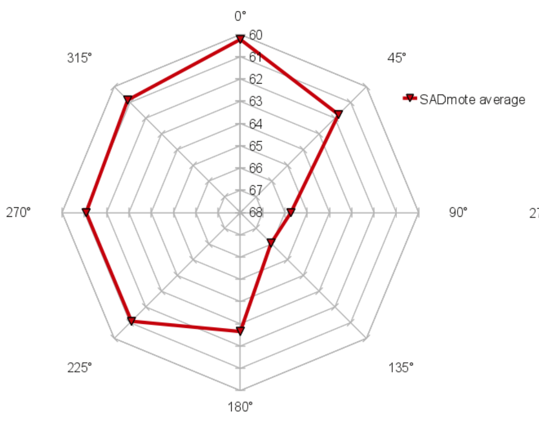


Figure 4.2: SADmote v02 packet delivery ratio (PDR) (top), RSSI (bottom left) and LQI (bottom right) in comparison with Tmote Sky

and DS2401P+ unique serial number chip, which is useful for automatic generation of a network address and unique identification of a mote. SADmote can be powered either from a programming board (also custom-built for this project) or two AA batteries.

There are several common components that were purposefully left out of SADmote in order to reduce its production cost and improve robustness, for example, on-board sensors, FTDI chip, LDO element, and USB connectors. To connect the mote to a programmer, either JTAG interface or SD-card compatible interface can be used. On mote's side, the SD card connector pads are simply plated on PCB. It was chosen as the simplest and most cost effective solution, not requiring any extra components.

External sensor boards can be attached to SADmote through expansion interface (analog and digital ports). A light sensor board was developed, specifically, two versions of it – one features Intersil ISL29003 sensor, the other Avago Technologies APDS9300. Both sensors measure irradiance (received energy level) are controlled through digital I^2C interface. The other sensors experimentally tested include SHT75 humidity and temperature sensor (digital interface) and a high-precision solar radiation sensor SQ-110 (analog interface) that measures photosynthetic photon flux density, i.e. the number of photons received in a given time on



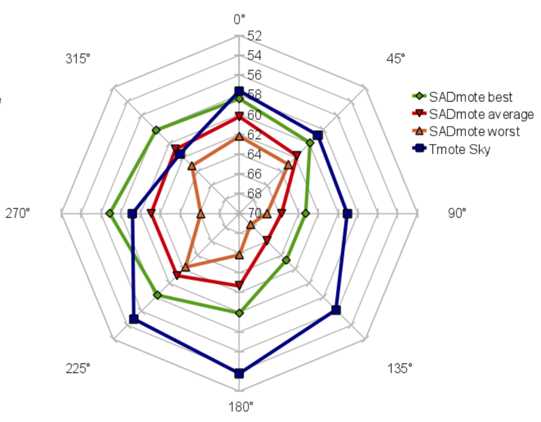


Figure 4.3: Measured SADmote v02 radiation pattern (left), in comparison with Tmote Sky (right)

a given area. Other sensors can be attached to SADmote as well, as soon as they are available – for example, soil humidity and temperature sensors, air chemical content sensors etc.

Our initial tests showed that in active mode SADmote v02 has energy consumption comparable to Tmote Sky (7.5 mW versus 6.9–8.4 mW); when radio is also turned on, then efficiency is slightly worse (76.5 mW versus 63.9 mW). SADmote v02 has radio communication range comparable to Tmote Sky: reliable line-of-sight links can be established in 150 meter distance and more. SADmote v03 has much greater radio rangle (more than 30 % of radio packets received at 1 km line-of-sight distance using external monopole antennas), partially due to increased maximal transmit power (10 dBm versus 0 dBm), and partially due to the longer wavelength being used.

The behavior of SADmote radio [Elsts 2012b] confirm the findings of [Ganesan 2002]. First, the received packet distribution has "long tail", i.e. the occasional packet on the occasional link is received at distances far exceeding the reliable communication range. Second, even at short distances no link is completely reliable; rather, received packet ratio from 70 % to 100 % is typical.

4.4 System architecture

The expected basic functionality of the system is the following:

- sensor motes measure microclimate parameters, store locally and transmit to the forwarder nodes;
- forwarder nodes forward data to base station using low-power wireless network;
- base station forwards data to a IEEE 802.11 wireless router;
- IEEE 802.11 wireless network transmits the data to a server;
- the server application parses the received data stream, performs consistency and validity checks, and stores the data in a database;

Table 4.2: Summary of SAD WSN deployments

Period	Activity
2009, 17 August – 15 September	Eight Tmote Sky motes and their supporting infrastructure (an outdoor router) deployed
2011, 19 September – 2 November	Twelve SADmote v02 with sensors, five Waspmotes and supporting infrastructure (server software, indoor and outdoor routers, roof antenna) deployed
2012, 16 April – 1 November	Nine SADmote v02 motes deployed as data loggers; from 30th August: 16 SADmote v03 motes with sensors, 2 SADmote v03 motes with directional antennas, and a single Zolertia Z1 mote deployed

- web interface makes the data accessible to authorized users anywhere on the Internet.

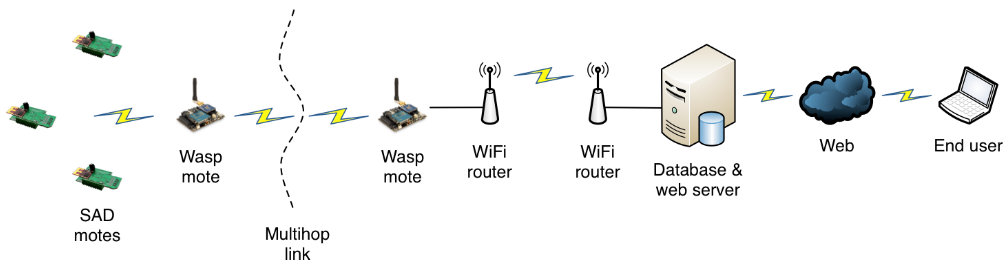


Figure 4.4: The conceptual architecture of SAD WSN

Taking into account the requirements, a hierarchical two-tier sensor network was designed for the SAD deployments (Fig. 4.4, Fig. 4.5). Tier one consisted of motes placed in the field and equipped with sensors (Fig. 4.6), which functioned as data sources. Tier two was formed by data forwarder motes with external (in some cases directional) antennas. Initially, static routing was used, but in 2012 deployment it was replaced by a self-configuring routing protocol. We exploited the fact that a weather station with external power was already present in the orchard, and put a IEEE 802.11 wireless router on its mast in order to reduce the number of forwarding hops required in the sensor network.

Waspmote [was 2012] devices were initially used for data forwarding as they had more powerful batteries (rechargeable, estimated 6600 mAh versus estimated 2700–2800 mAh of AA batteries used by SADmotes), and featured more powerful radio modules with external antennas. However, in the 2011 deployment we discovered that the larger batteries does not compensate the larger energy consumption (measured 3.07 milliwats even in the sleep mode, compared to tens of microwatts on SADmotes), and the field lifetime is quite short. Therefore in 2012 all of the network was formed by SADmotes, except base station, for which Zolertia Z1 was used.

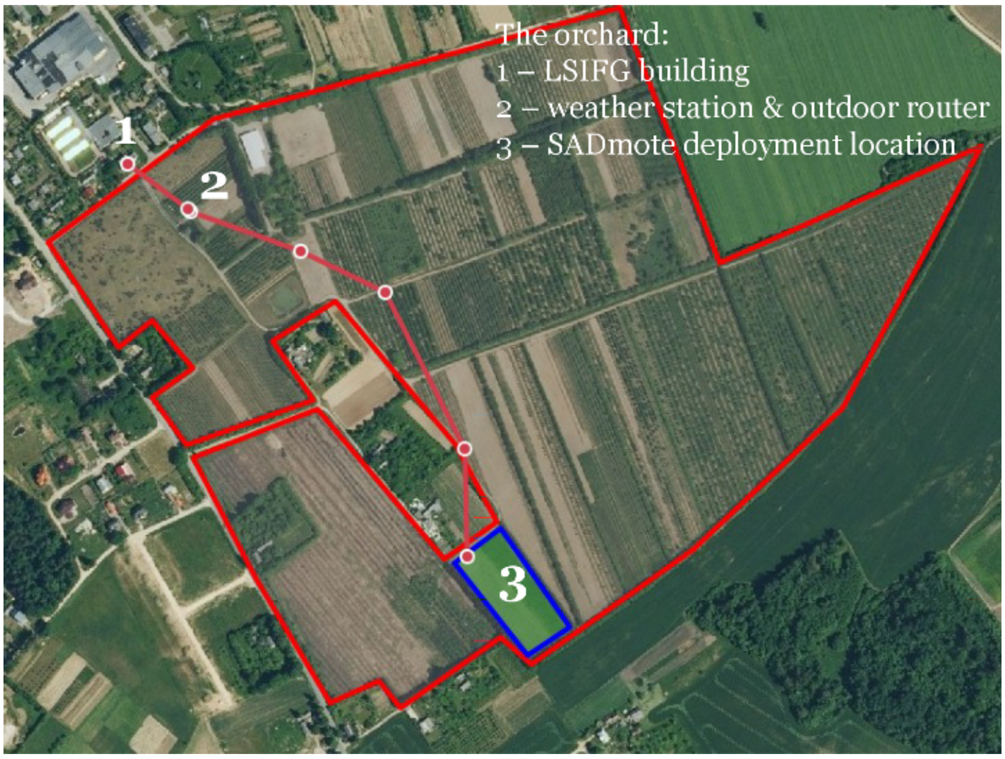

The orchard:
1 – LSIFG building
2 – weather station & outdoor router
3 – SADmote deployment location
1
2
3

Figure 4.5: The structure and location of SAD WSN in 2011. The points in the middle mark repeater and base station locations

The data gathering motes were placed from 0.1 m to 2 m above ground. The forwarding motes and the base station were placed from 3 m to 8 m above ground depending on the position. The terrain of the orchard, as well as several intermediate rows of trees occluded the lines-of-sight for wireless communications. At least the Fresnel zone (for 800 MHz frequency band) was always obstructed.

MansOS was used for the software, which provided a number of useful features:

- Routing (collection-tree) protocol with built-in millisecond-precision time synchronization;
- A TDMA MAC protocol with four mote roles (data originator, data collector, data forwarder, and base station);
- Accurate software-based current time estimation even when a mote becomes severed from the network (Section 2.5.2);
- Relatively simple API that allowed to to program the SAD application (separate for each mote role) in 200–300 lines of code in 2011;
- Last but not least, SEAL integration, that allowed to program the SAD application (for whole network all mote roles at once) in only 14 lines of code in 2012.

Over-the-air reprogramming is a MansOS feature in development (Section 2.5.1.3) that may be utilized in the future for this application.

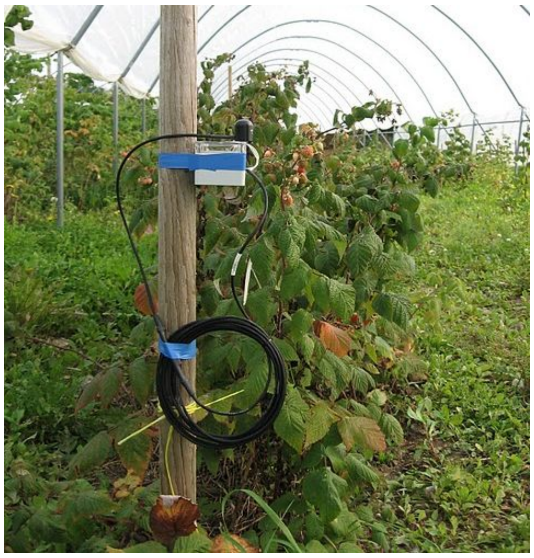


Figure 4.6: A SADmote in raspberries

The SAD application code (using SEAL) is given in Listing 3.22 and analyzed in Section 3.5.3.2.

4.5 Lessons learned

The deployments in 2011 and 2012 were able to gather number of microclimate measurements. In 2011 the sensor devices recorded 50 006 measurements (using 5 minute interval), in 2012: 140 267 measurements (majority of them was recorded using 10 minute interval, the rest: using 5 minute interval). Nevertheless, the deployments were plagued with number of problems, as well as missing features. Here we analyze these problems and show how they were solved.

Data quality. M. Welsh [Welsh 2010] describes his own experience with a sensor network deployment (also described in [Werner-Allen 2006]) and stresses the fact that initially in their deployment "data quality was not what the real scientists expected". He names three aspects of the problem: "missing samples", "bad timing", and "uncalibrated readings". Our experience matches not only his observation that achieving research-grade data quality is hard, but also precisely matches the three aspects of this problem!

- **Missing samples.** In the 2009 pilot deployment, missing samples were an issue because of the unreliability of radio communication. In subsequent deployments the data was also stored locally. Even so, the deployments also experienced this problems, because of shorter-than-expected battery lifetime. The reasons: software failing to enter low-power mode, and failures because of rain and dew getting into the device enclosures, and causing hardware to fail. At the moment both of these problems have been fixed.

- **Bad timing.** The potential problems with timestamps that are off were initially handled by developing a routing protocol that includes time synchronization. It turned out that it was insufficient in case where the network became partitioned and the motes were left "on their own". In order to keep time accounting accurate even in this case, we implemented advanced timing correction in MansOS (Section 2.5.2).
- **Uncalibrated readings.** There are three incompatible light measurement methods: measuring the perceived power of light (expressed in luxes), measuring the energy of light (in $\mathrm{W\,m^{-2}}$), and measuring the quantum flux of the light (in $\mu\mathrm{mol\,m^{-2}\,s^{-1}}$). The perceived power of the light (luxes) are related to the physiology of human eye and cannot be used to measure the radiation usable by plants. From the two others, quantum flux value is most directly correlated to the amount of energy that plants can extract in photosynthesis process. This is because the energy a plant can use from light waves does *not* increase as the wavelength decreases beyond 700 nm wavelength. Shorter-wave length light is going to have more energy (when $\mathrm{W\,m^{-2}}$ are measured) while having the same amount of energy that is usable for photosynthesis.

  Initially the environmental scientists wanted light readings in luxes, as this measurement unit was familiar to them; they suggested using a luxometer for sensor calibration. We tried to correct their understanding and offered reading both the energy and quantum flux, and optionally converting between them after applying spectral correction to the measured energy values.

  There was also the problem by different light values registered by different sensors of a single kind, which could be solved by calibrating the sensors and applying postprocessing to the data.

Extending the radio range. High number of intermediate hops has the drawbacks of both decreased packet delivery rate and higher energy consumption. Taking this in account, one of our objectives was to reduce the number of intermediate hops needed, by increasing motes' radio range and radio link stability. We saw a number of ways how it would be achieved:

- Use directional antennas for some of the intermediate hops, or at least for the base station placed next to the router. The motes in the middle would benefit from dual-beam antennas. Taking in account that the forwarding path is not always a straight line, the beams of the antenna should be apart not for 180 degrees, but a deployment-specific number of degrees. The construction of such antennas is an interesting engineering challenge in itself.
- Use lower radio frequencies. Sub-1GHz electromagnetic waves are not absorbed by foliage and other small obstacles as much as 2.4 GHz waves, as they have longer wavelengths. For example, assuming a 200 m distance between radio receivers, 100 m of foliage within the line-of-sight path, and using

Weissenberg's modified exponential decay model [Seybold 2005] one can calculate path loss using the formula:

$$Loss(dB) = 1.33F^{0.284}d^{0.588}$$

where F is frequency in GHz, and d is foliage depth in meters. The resulting loss is 25.73 dB for 2.450 GHz frequency, and 19.16 dB for 0.868 GHz – a difference of 6.57 dB, allowing to approximately *double* the communication distance.

- Place repeater motes higher above ground. For this purpose, some of the higher trees located in the orchard, large poles or small towers placed there could be used. However, due to elevated ground in the middle of the deployment site, a single line-of-sight radio link reaching the farthest parts of the orchard does not appear to be feasible, at least not without mayor investments in tower construction.

The first two options were implemented for the 2012 deployment, leading to reduced number of intermediate hops (two hops in 2012 versus four hops in 2011).

Avoiding a single point-of-failure. Having a single forwarding path in the network means that a single failed node can cause the data to no longer be accessible online. As our experience with failing Waspmotes shows, this problem should be taken in account for network design considerations. Fault-tolerance is especially important if sensor readings are used to generate alerts, or to activate orchard watering or other control systems. The solutions are:

- Place repeater motes more frequently, so that each mote can see not only the nearest neighbor in both directions, but one or two next neighbors as well. In this case, the path is still functional even if some of the motes fail, but then we have to cope with the previously described drawbacks of having too many repeater motes, as well as more radio collisions.
- Construct two parallel forwarding paths. To avoid interference, non-intersecting radio channels should be used. This is a robust solution, although relatively expensive and labor-intensive to set up, as twice as many motes and mounting locations are required. It would benefit from an advanced multipath routing protocol, which would provide a kind of load-balancing, thus increasing the lifetime of the motes by decreasing their energy expenditure.

Our research group plan to implement the first option for 2013 deployment.

Unit testing each component. Before the deployment in 2011 the author performed unit testing on each mote by uploading a specifically designed MansOS SAD testing application and checking whether all the components are working. Specifically, the software checked light and humidity sensor output, radio transmission and reception, writing in and reading from external flash, and other hardware components on SADmote. In the deployment our research group learned both the value of these tests and that our testing was incomplete. It turned out that most of the motes deployed were unable to enter low-power modes correctly. No one can know whether all of them had this defect initially; in any case, when testing afterwards, only one from all of the motes deployed was able to enter this mode. Since energy consumption in active mode is much higher than consumption in sleep mode (current draw is 2.5 mA in active mode versus tens of microamperes in sleep mode), and a normally functioning mote ought to spend most of the time in this sleep mode (as much as 99.9%, the measurement and data transmission takes approximately 0.3 seconds and is performed once in every five minutes), the consequences on long-term energy efficiency are dramatic. These 2011 test deployments were too short, but a quick calculation shows that the motes would certainly fail in a deployment that would last at least for two months. Assuming battery capacity 2800 mAh, operating voltage of 3.0 V, and average current consumption of 2.8 mA (active mode with the occasional radio packet transmission and reception), the result is 1000 hour lifetime from a pair of batteries, which corresponds to approximately 41 days. Consequently, in 2012 unit testing was extended with low-power mode testing, which allowed to solve this problem.

System level testing. Unit testing is useful, but not sufficient to discover all the problems that can manifest themselves only in real-world settings. For example, expected radio communication quality cannot be reliably modelled from based only on data from a few site surveys. A survey helps in modelling the topographic relief of the site with high precision, but other factors like precipitation, presence or absence of snow cover, vegetation and leaf growth are season-dependent or occasional, therefore cannot be discovered by short-term studies. Nevertheless, they have tremendous influence on radio signal propagation as well [Seybold 2005].

In the deployments, our research group encountered both problems that could be discovered by more careful system level testing in-house, and problems that apparently could not.

First, it was the failure of the Libelium Meshlium router after trying to reconfigure it. Eventually the router had to be sent for warranty repair back to the manufacturer, which took almost two months, re-engineer the WiFi part of the network and use a MikroTik router instead, which significantly delayed our deployment.

Second, some of the SADmote v02 motes exhibited tendency to hang when radio was actively used. The author had not encountered this problem in in-house testing because transmit and receive modes were usually tested separately, while this problem manifests itself only when packet transmissions are mixed packet receptions, which are common in real-world settings.

Another problem that the author suspects was triggered by radio packet reception is the constant rebooting of Waspmotes. It is possible that our Waspmote software had some errors, but we note that the examples provided by Libelium were used as the basis of our code. Because of this problem, the forwarding path was never reliable in 2011 and completely stopped working after only a few days. The motes also soon ran out of batteries.

In 2012 the design of a new version of SADmote changed the radio chip (as described in Section 4.3), and the radio stability problem was solved.

Closing the control loop. A natural extension of the SAD WSN design is to close the control loop and use the network not only to monitor the microclimate, but also to control the watering and other systems in the field. So far this has not been a typical use of sensor networks due to the experimental status and perceived instability of this technology, although some projects with positive real-world experience exist [Ceriotti 2011]. We consider such an extension to be a future research goal. The first step in implementation of such a system would be an alarm signal in case some microclimate parameters reach critical values. Such an extension would still require a human operator in the loop, but would provide a significant assistance in detection of such critical events, for example, frost.

Mechanisms for run time control and assurance. Domain scientists, having no previous WSN experience, no programming skills and lack of intent to setup complicated development environments on their computers, require an intuitive visual feedback on the way network functions. Current version of SADmote features a single LED, initially intended for debugging purposes. As it turns out, LEDs are useful for run-time assurance as well. In the end of our experiment some of the motes apparently had frozen when trying to read their humidity sensors, which were damaged because of their incomplete protection from precipitation. Since our software turns on LED during the sensor reading activity, one can quickly tell which motes are not functioning anymore – their LED is constantly either on or off. The domain scientists suggested extending this approach and use multiple LEDs, each corresponding to a specific sensor. For example, the mote could have red, green and blue LEDs, and each of these was associated with a specific sensor. Glowing red LED would mean that light sensor has failed; blue LED: humidity sensor, and so on.

Run-time reprogramming. MansOS supports over-the-air reprogramming; however, this feature is in experimental status. It was not included it in these initial tests, as firstly, we did not want to compromise the initial deployment of our sensor network by software bugs, and secondly, we wanted to have a stable minimal base of the software tested in field conditions, useful as a benchmark for later tests, when more features will be added and the unavoidable software bugs start to creep out.

The deployments convinced us about the necessity of such a feature even in medium size networks. At the first inspection of the first network (7th October, 2011) our research group discovered that the motes are not functioning correctly and need to be updated. Afterwards some of them were reprogrammed. Reprogramming a mote meant that it should be not only physically accessed, but also taken out of the box, attached to the programming board, which in turn is attached to a laptop, and so on. As a consequence, reprogramming just nine motes (eight successfully, one could not be programmed at that time) took two hours (from 18:00 till 20:00 that evening). Even if this procedure took just five minutes for each mote, a network from 66 motes (our target size for the 2013 deployment) would require more than five hours for a single person to reprogram! Hence, support for reprogramming the motes in a non-interactive mode without physical access is essential.

Usability aspects of WSN applications. Even more, reprogramming the motes – updating software versions, reconfiguring sensor reading frequency and other parameters – should be easy enough for agro-scientists to do on their own, without the help of computer professionals. Understanding of this, as well as the observation that a typical applications are conceptually simple. Both were the impulses that led to the creation of SEAL. By working close to the practitioners, the author was able to envision software that can be more usable and easier to learn for them compared with low-level APIs, and more flexible compared with application-specific configuration interfaces.

4.6 Concluding remarks

The design of a WSN system is an iterative and incremental process. Deploying a sensor network which can consist of tens of motes is a significant engineering challenge, being ridden with unexpected problems [Langendoen 2006]. Nevertheless, our deployments were able to gather data for several months.

Deploying wireless sensor networks is still a global research problem. The author hopes that the lessons his research group has learned will be useful to other prospective WSN designers and users, either in precision agriculture or in other areas. To the best of our knowledge, SAD (in years 2009 and 2011) was the first time a *wireless* sensor network was used in Latvia for environmental monitoring in precision agriculture. Therefore, our work helps by discovering the specific local conditions & problems and investigating the specific requirements.

At the moment there are two main outcomes of the still ongoing SAD project. The first is the custom sensor device, SADmote, designed specifically for this application. We wanted the adaptability, simplicity and efficiency offered by a application-specific hardware design. SADmote has fewer components than state-of-art WSN motes such as Tmote Sky, therefore the production cost is lower and the number of potential failure points are limited. On the other hand, it has a micro-SD card slot, allowing to store much larger quantities of data. Also, it has a radio with higher communication distance.

The second set of outcomes is related to the role that SAD played in the development of our software. We realized the importance of real-world validation, as well as the importance of feedback from users. As a result, SAD project largely directed the development of SEAL and MansOS in the last few years (starting from 2011 summer). SAD has helped us understand and specify the requirements of WSN software, as well as to evaluate our software in field condition (SEAL in 2012 deployment, MansOS in all deployments).

Chapter 5

Conclusion

Contents

5.1 Results

The main results of the dissertation are a number of approaches and tools that facilitate wireless sensor network application development, including:

- SEAL, a domain-specific language suitable for development of wireless sensor network applications, with relatively short grammar, and with concise syntax (Chapter 3);
- the compiler of this domain-specific language, which supports automatic inclusion of low-level operations in the binary code image, for example, automatic usage of low-power modes, or numerical constant scaling in order to map them between physical units and sensor measurement domains (Section 3.3);
- visual programming language SEAL-Blockly, and visual development environment suitable for sensor network application development (Section 3.3.5);
- the idea and implementation of a semi-automatic software component selection mechanism for sensor network operating systems that are implemented in C programming language (Section 2.3);
- approach for millisecond-accuracy time accounting in wireless sensor network operating systems (Section 2.5.2);
- approach for reducing the amount of platform-dependent code in wireless sensor network operating systems by using four-layer code architecture (Section 2.2).

The results are integrated in a single framework. Even though most the results at this point are implemented in a specific operating system MansOS and in a specific programming language SEAL, they can be applied to other suitable contexts as well, for example, integrated in other sensor network operating systems. Some initial steps already have been taken in this direction (Chapter 2) and some of the results (the component selection mechanism and the code architecture) have been partially adopted in Contiki with good results (more than 12 % reduction in both RAM and code memory usage in the first case, and more than 17 % reduction in source code size in the second case).

The central theses that were put forward in the Introduction chapter are supported by these results. The analysis of the existing sensor network programming approaches show that they already implement many ideas appropriate for development of wireless sensor network applications, but frequently at the cost of being not straightforward to use, and, especially, not straightforward to start using. Therefore their accessibility for novice programmers should be increased. Chapter 3 details our approach to this: creating a custom sensor network application development language. The language is compared with several other DSL and general purpose programming languages, shown to have smaller syntax & grammar, and shorter & less complex application source code. The integrated development environment helps by allowing to start application prototyping (i.e. to get visible results) faster; the visual programming language and environment helps by providing set of predefined blocks, therefore syntax errors are impossible, and load on developer's working memory is reduced.

The component selection mechanism (Section 2.3) turned out to be surprisingly versatile and powerful tool that allows to: (1) modularize the operating system by allowing to exchange components used in an application without changing its source code; (2) extend the operating system in application-specific or platform-specific ways without modifying the core of it; (3) optimize the operating system and applications by pruning unused components from the final binary code image.

Our framework at the system level offers both preemptive multithreading and event-based execution out of the box. The application developer can choose the execution mode she prefers for each particular application. The thread-based concurrency model is likely to be more familiar, the event-based – more efficient. Another notable system-level features are static memory allocation and parametrized hardware interfaces; they are adopted from TinyOS, but are implemented in a way that makes them accessible from C rather than nesC code, so system programmers without WSN experience can use them without learning an additional programming language. Finally, there is the millisecond abstraction (used for timers and time accounting) that is familiar to everyone, but so far has lacked an accurate implementation on several popular WSN platforms.

5.2 Conclusions

1. Implementation of sensor network applications in a domain specific programming language allows the code to be much more concise and more free from low-level details compared to implementation of the same applications in C, while not necessarily leading to loss of efficiency. For example, for the non-

empty applications analyzed in this thesis, the resource overhead of SEAL does not exceed 40 % (which, for comparison, is smaller than the resource usage overhead for MansOS C applications than enable multithreading).

2. Implementation of sensor network applications in a domain specific programming language allows to use concepts that are either already familiar to the users or are easy to grasp, such as concepts with concrete real-world counterparts. For example, SEAL allows to use components that correspond to physical sensors and constants expressed in generally-known units of physical measurements.

3. A domain-specific language with high level of abstraction can be easily adapted for visual development environments, as our experience with SEAL / MansOS IDE and SEAL-Blockly IDE shows.

4. Semi-automatic component selection mechanism allows the user to easily extend, modularize and optimize both the sensor network operating system and its applications, leading to executables with smaller code memory and RAM usage footprint.

5. The code architecture of a wireless sensor network operating system has significant impact on the amount of platform-specific code, therefore also on its portability. For example, by fully separating *architecture*-specific code, the amount of *platform*-specific code can be reduced.

6. The state-of-art wireless sensor network operating systems lack such an obvious functionality as millisecond-accuracy time accounting, even though the hardware allows to implement it without great loss of efficiency.

7. Evaluation of software in a real-world use case not only allows to test the software, but also creates preconditions for better understanding of real-world issues (such as radio signal absorption in tree foliage) and requirements (such as data quality requirements) in wireless sensor networks.

5.3 List of publications

The results of this thesis have been included in a number of scientific publications, listed below. The total number of publications is 13 (including short papers and poster abstracts). Five of them have been indexed in SCOPUS database, and at least two more are expected. The author personally has contributed at least 70 % of text in every publication in which he is named as the first among respective collaborating authors.

Publications related to SEAL:

• **A. Elsts** and L. Selavo, *"A User-Centric Approach to Wireless Sensor Network Programming Languages,"* in Proceedings of the Third International Workshop on

Software Engineering for Sensor Network Applications (SESENA 12), pp. 29–30, ACM, 2012 [Elsts 2012d], **indexed in SCOPUS**.

• **A. Elsts**, J. Judvaitis, and L. Selavo, *"Poster Abstract: SEAL: An Easy-to-use Sensor Node Application Development System,"* in Poster and Demo Proceedings of 9th European Conference on Wireless Sensor Networks (EWSN 2012), pp. 31–32, Trento, Italy, 2012 [Elsts 2012c].

• J. Judvaitis, **A. Elsts**, and L. Selavo, *"Demo Abstract: SEAL-Blockly: Sensor Network Visual Programming Using a Web Browser,"* in Poster and Demo Proceedings of 10th European Conference on Wireless Sensor Networks (EWSN 2013), 4 pp., 2013 [Judvaitis 2013].

• **A. Elsts**, J. Judvaitis, and L. Selavo, *"SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers,"* in Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2013), pp. 220–227, IEEE, 2013 [Elsts 2013a], expected to be indexed in SCOPUS.

Publications related to MansOS:

• G. Strazdins, **A. Elsts**, and L. Selavo, *"MansOS: easy to use, portable and resource efficient operating system for networked embedded devices,"* in Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys'10), ACM, New York, NY, USA, pp. 427–428, 2010 [Strazdins 2010], **indexed in SCOPUS**.

• **A. Elsts**, G. Strazdins, A. Vihrov, and L. Selavo, *"Design and Implementation of MansOS: a Wireless Sensor Network Operating System,"* Scientific Papers, University of Latvia, pp. 79–105, 2012 [Elsts 2012e].

• G. Strazdins, **A. Elsts**, K. Nesenbergs and L. Selavo, *"Wireless Sensor Network Operating System Design Rules Based on Real World Deployment Survey,"* Journal of Sensor and Actuator Networks, pp. 509–556, vol. 2(3), ISSN 2224-2708, MPDI, 2013 [Strazdins 2013].

• **A. Elsts** and L. Selavo, *"Improving the Usability of Wireless Sensor Network Operating Systems,"* Position Papers of the Federated Conference On Computer Science and Information Systems 2013 (FedCSIS 2013), pp. 89–94, Polskie Towarzystwo Informatyczne, 2013 [Elsts 2013b].

Publications related to applications of MansOS and SEAL:

• R. Zviedris, **A. Elsts**, G. Strazdins, A. Mednis, and L. Selavo, *"LynxNet: Wild Animal Monitoring Using Sensor Networks,"* Lecture Notes in Computer Science, 2010, Volume 6511, Real-World Wireless Sensor Networks (REALWSN'10), pp. 170–173, Springer, 2010 [Zviedris 2010], **indexed in SCOPUS**.

• **A. Elsts**, R. Balass, J. Judvaitis, and L. Selavo, *"SAD: Wireless Sensor Network System for Microclimate Monitoring in Precision Agriculture,"* in Proceedings of the 5-th international scientific conference Applied information and communication technologies (AICT 2012), Jelgava, Latvia, April 26-27, ISBN 978-9984-48-065-7, pp. 271–281, 2012 [Elsts 2012a].

• **A. Elsts**, R. Balass, J. Judvaitis, R. Zviedris, G. Strazdins, A. Mednis, and L. Selavo, *"SADmote: A Robust and Cost-Effective Device for Environmental Monitoring,"* in Proceedings of the conference on Architecture of Computing Systems (ARCS 2012), pp. 225–237, Springer-Verlag Berlin Heidelberg, 2012 [Elsts 2012b], **indexed in SCOPUS**.

• A. Mednis, **A. Elsts**, and L. Selavo, *"Embedded Solution for Road Condition Monitoring Using Vehicular Sensor Networks,"* in proceedings of the 6th International Conference on Application of Information and Communication Technologies (AICT2012), pp. 1–5, Georgia, Tbilisi, IEEE, 2012 [Mednis 2012b], **indexed in SCOPUS**.

• **A. Elsts**, A. Mednis, and L. Selavo, *"Bayesian Network Approach to Vehicle Mode Monitoring Using Embedded System with 3-axis Accelerometer,"* in Special Issue on Artificial Intelligence in Robotics and Imaging of International Journal of Imaging & Robotics, pp. 67–80, vol. 12(1), ISSN 2231-525X, CESER Publications, 2014 [Elsts 2014], expected to be indexed in SCOPUS.

The author has presented the results in a number of international scientific conferences:

Oral presentations:

• *"A User-Centric Approach to Wireless Sensor Network Programming Languages,"* Third International Workshop on Software Engineering for Sensor Network Applications (SESENA 12), Zurich, Switzerland, June 2, 2012.

• *"SADmote: A Robust and Cost-Effective Device for Environmental Monitoring,"* Architecture of Computing Systems (ARCS 2012), Garching, TU Muenchen, Germany, February 28–March 02, 2012.

• *"SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers,"* 39th EUROMICRO Conference on Software Engineering and Advanced Applications, Santander, Spain, September 4–6, 2013.

• *"Improving the Usability of Wireless Sensor Network Operating Systems,"* Federated Conference On Computer Science and Information Systems 2013 (FedCSIS 2013), Krakow, Poland, September 8–11, 2013.

Poster & demo presentations:

• *"MansOS: easy to use, portable and resource efficient operating system for networked embedded devices,"* 8th ACM Conference on Embedded Networked Sensor Systems (SenSys'10), Zurich, Switzerland, November 3–5, 2010.

• *"SEAL: An Easy-to-use Sensor Node Application Development System,"* 9th European Conference on Wireless Sensor Networks (EWSN 2012), Trento, Italy, February 14–17, 2012.

• *"SEAL-Blockly: Sensor Network Visual Programming Using a Web Browser,"* 10th European Conference on Wireless Sensor Networks (EWSN 2012), Ghent, Belgium, February 13–15, 2013.

The author has also presented the results in a number in local scientific conferences and seminars, including the 68th Scientific Conference of the University of Latvia, 2010 (presentation *"SAD – bezvadu sensoru tīkls augļu dārzā"*, the 70th Scientific Conference of the University of Latvia, 2012 (*"SADmote: A Robust and Budget-Efficient Device for Environmental Monitoring"*), the First Smart Sensors seminar, 2010 (*"MansOS: operētājsistēma sensoru tīkliem un iegultajām ierīcēm"*), the Third Smart Sensors seminar, 2011 (*"SeAdScript: deklaratīvā sensoru programmēšana nespeciālistiem"*), the Fifth Smart Sensors seminar, 2012 (*"Praktiskas sarunas cilvēkiem ar viediem sensoriem valodā SEAL"*), and the Second University of Latvia and *Latvijas mobilais telefons* computer science days, 2012 (*"Bezvadu sensoru tīklu programmēšana nespeciālistiem"*).

One dissertation-related bachelor's thesis was supervised by the author. The thesis, *"Visual wireless sensor network programming environment"* [Judvaitis 2012] was successfully defended (grade 10 out of 10) by Janis Judvaitis at Faculty of Computing, University of Latvia, and was selected for the national IT student thesis contest in 2012.

Bibliography

[Abran 2003] A. Abran, A. Khelifi, W. Suryn and A. Seffah. *Usability meanings and interpretations in ISO standards.* Software Quality Journal, vol. 11, no. 4, pages 325–338, 2003. (Cited on page 10.)

[AdvanticSYS 2013] AdvanticSYS. *XM1000.* `http://www.advanticsys.com/wiki/index.php?title=XM1000`, 2013. [Online: accessed 26.04.2013.]. (Cited on pages 8, 39 and 65.)

[Alberts 2013] M. Alberts, U. Grinbergs, D. Kreismane, A. Kalejs, A. Dzerve, V. Jekabsons, N. Veselis, V. Zotovs, L. Brikmane and B. Tikuma. *New wireless sensor network technology for precision agriculture.* In Proceedings of the 6-th international scientific conference Applied information and communication technologies (AICT 2013), pages 77–83, 2013. (Cited on page 158.)

[Arduino Team 2012] Arduino Team. *Arduino.* `http://arduino.cc/`, 2012. [Online; accessed 30.03.2013.]. (Cited on pages 24 and 159.)

[ARM Holdings 2013] ARM Holdings. *Cortex-M0+ Processor.* `http://www.arm.com/products/processors/cortex-m/cortex-m0plus.php`, 2013. [Online: accessed 04.04.2013.]. (Cited on page 8.)

[atm 2011] *Atmel ATmega328P.* `http://www.atmel.com/dyn/products/product_card.asp?part_id=4198`, 2011. [Online; accessed 30.09.2011.]. (Cited on page 8.)

[Atmel Corporation 2013] Atmel Corporation. *Atmel AVR 8- and 32-bit Microcontrollers.* `http://www.atmel.com/products/microcontrollers/avr/default.aspx`, 2013. [Online: accessed 04.04.2013.]. (Cited on page 8.)

[Avilés-López 2009] E. Avilés-López and J.A. García-Macías. *TinySOA: a service-oriented architecture for wireless sensor networks.* Service Oriented Computing and Applications, vol. 3, no. 2, pages 99–108, 2009. (Cited on pages 32 and 128.)

[Baggio 2005] A. Baggio. *Wireless sensor networks in precision agriculture.* In ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), Stockholm, Sweden. Citeseer, 2005. (Cited on page 158.)

[Bai 2009] L.S. Bai, R.P. Dick and P.A. Dinda. *Archetype-based design: Sensor network programming for application experts, not just programming experts.* In Proceedings of the 2009 International Conference on Information Processing in Sensor Networks, pages 85–96. IEEE Computer Society, 2009. (Cited on pages 11, 33, 34, 97, 100, 101, 127, 128, 138, 139 and 155.)

[Bakshi 2005] A. Bakshi, V.K. Prasanna, J. Reich and D. Larner. *The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems.* In Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services, pages 19–24. USENIX Association, 2005. (Cited on page 28.)

[Barnum 2010] Carol M Barnum. Usability testing essentials: ready, set... test. Morgan Kaufmann, 2010. ISBN 978-0123750921, 408 pages. (Cited on pages 90 and 152.)

[Baums 2007] A. Baums, N. Zaznova and V. Redjko. *Augu māju temperatūras un relatīvā mitruma monitoringa sistēmu izstrāde.* Rīgas Tehniskās universitātes zinātniskie raksti, vol. 32, pages 77–83, 2007. (Cited on page 158.)

[Bhatti 2005] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson and R. Han. *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms.* Mobile Networks and Applications, vol. 10, no. 4, pages 563–579, 2005. (Cited on pages 19, 20, 22, 36 and 54.)

[blo 2012] *Blockly: A visual programming editor.* `http://code.google.com/p/blockly`, 2012. [Online: accessed 01.10.2012.]. (Cited on page 126.)

[Brock 2009] J.D. Brock, R.F. Bruce and S.L. Reiser. *Using Arduino for introductory programming courses.* Journal of Computing Sciences in Colleges, vol. 25, no. 2, pages 129–130, 2009. (Cited on pages 8, 20 and 24.)

[Brooks 1987] Frederick P. Brooks Jr. *No Silver Bullet: Essence and Accidents of Software Engineering.* Computer, vol. 20, no. 4, pages 10–19, April 1987. (Cited on pages 1 and 16.)

[BusinessWire 2012] BusinessWire. *Research and Markets: Wireless Sensor Networks 2011-2021.* `http://www.businesswire.com/news/home/20120217005293/en/Research-Markets-Wireless-Sensor-Networks-2011-2021`, 2012. [Online: accessed 08.05.2013.]. (Cited on page 1.)

[c-i 1999] *ISO/IEC 9899:1999 – Programming languages – C*, 1999. (Cited on pages viii, 131 and 132.)

[Cao 2008] Q. Cao, T. Abdelzaher, J. Stankovic and T. He. *The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks.* In IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on pages 20, 23, 24, 34, 36 and 78.)

[Cardell-Oliver 2013] Rachel Cardell-Oliver, Stefan Böttcher and Christof Hübner. *Data-Aware, resource-aware, lossless compression for sensor networks.* In

EWSN'13 Proceedings of the 10th European conference on Wireless Sensor Networks, pages 83–98. Springer, 2013. (Cited on page 138.)

[Casati 2012] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, PM Montera, L. Mottola, F.J. Oppermann, G.P. Picco*et al.* *Towards business processes orchestrating the physical enterprise with wireless sensor networks.* In Software Engineering (ICSE), 2012 34th International Conference on, pages 1357–1360. IEEE, 2012. (Cited on pages 6, 29 and 127.)

[Ceriotti 2011] M. Ceriotti, M. Corrà, L. D'Orazio, R. Doriguzzi, D. Facchin, S.T. Guna, G.P. Jesi, R.L. Cigno, L. Mottola, A.L. Murphy*et al.* *Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels.* In Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on, pages 187–198. IEEE, 2011. (Cited on pages 139, 140, 144 and 170.)

[Chu 2006] D. Chu, A. Tavakoli, L. Popa and J. Hellerstein. *Entirely declarative sensor network systems.* In Proceedings of the 32nd international conference on Very large data bases, pages 1203–1206. VLDB Endowment, 2006. (Cited on pages 31 and 97.)

[Chu 2007] D. Chu, L. Popa, A. Tavakoli, J.M. Hellerstein, P. Levis, S. Shenker and I. Stoica. *The design and implementation of a declarative sensor network system.* In Proceedings of the 5th international conference on Embedded networked sensor systems, pages 175–188. ACM, 2007. (Cited on pages 6 and 31.)

[Clark 1997] G. Clark. *Telnet Com Port Control Option.* RFC 2217, Internet Engineering Task Force, October 1997. (Cited on page 62.)

[Corke 2012] Peter Corke. *Environmental wireless sensor networks: a decade's journey from the lab to the field.* Keynote speech at the 9th European Conference on Wireless Sensor Networks (EWSN'12), 2012. (Cited on page 2.)

[Costa 2007] P. Costa, L. Mottola, A. Murphy and G. Picco. *Programming wireless sensor networks with the teeny lime middleware.* Middleware 2007, pages 429–449, 2007. (Cited on page 32.)

[Detweiler 2010] C. Detweiler, M. Doniec, M. Jiang, M. Schwager, R. Chen and D. Rus. *Adaptive decentralized control of underwater sensor networks for modeling underwater phenomena.* In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, pages 253–266. ACM, 2010. (Cited on page 160.)

[Duffy 2008] C. Duffy, U. Roedig, J. Herbert and C. Sreenan. *A Comprehensive Experimental Comparison of Event Driven and Multi-Threaded Sensor Node*

Operating Systems. Journal of Networks, vol. 3, no. 3, pages 57–70, 2008. (Cited on pages 19, 20, 52 and 85.)

[Dunkels 2004] A. Dunkels, B. Grönvall and T. Voigt. *Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors.* In Proceedings of the Annual IEEE Conference on Local Computer Networks, pages 455–462, Los Alamitos, CA, USA, 2004. IEEE Computer Society. (Cited on pages 22 and 36.)

[Dunkels 2006] A. Dunkels, O. Schmidt, T. Voigt and M. Ali. *Protothreads: simplifying event-driven programming of memory-constrained embedded systems.* In Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM. (Cited on page 22.)

[Dyo 2010] V. Dyo, S.A. Ellwood, D.W. Macdonald, A. Markham, C. Mascolo, B. Pásztor, S. Scellato, N. Trigoni, R. Wohlers and K. Yousef. *Evolution and sustainability of a wildlife monitoring sensor network.* In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10, pages 127–140, New York, NY, USA, 2010. ACM. (Cited on page 160.)

[Elsts 2012a] A. Elsts, R. Balass, J. Judvaitis and L. Selavo. *SAD: Wireless Sensor Network System for Microclimate Monitoring in Precision Agriculture.* In Proceedings of the 5-th international scientific conference Applied information and communication technologies (AICT 2012), pages 271–281, 2012. (Cited on pages 140, 157 and 175.)

[Elsts 2012b] A. Elsts, R. Balass, J. Judvaitis, R. Zviedris, G. Strazdins, A. Mednis and L. Selavo. *SADmote: A Robust and Cost-Effective Device for Environmental Monitoring.* In A. Herkersdorf, K. Römer and U. Brinkschulte, editeurs, Architecture of Computing Systems ARCS 2012, volume 7179 of *Lecture Notes in Computer Science*, pages 225–237. Springer Berlin / Heidelberg, 2012. (Cited on pages 157, 161, 163 and 176.)

[Elsts 2012c] A. Elsts, J. Judvaitis and L. Selavo. *Poster Abstract: SEAL: An Easy-to-use Sensor Node Application Development System.* In Poster and Demo Proceedings of 9th European Conference on Wireless Sensor Networks (EWSN 2012), pages 31–32, 2012. (Cited on page 175.)

[Elsts 2012d] A. Elsts and L. Selavo. *A User-Centric Approach to Wireless Sensor Network Programming Languages.* In SESENA '12: Proceedings of the 3rd Workshop on Software Engineering for Sensor Network Applications, pages 29–30, New York, NY, USA, 2012. (Cited on pages 11, 98, 116 and 175.)

[Elsts 2012e] A. Elsts, G. Strazdins, A. Vihrov and L. Selavo. *Design and Implementation of MansOS: a Wireless Sensor Network Operating System.* Acta

Universitatis Latviensis. Computer Science and Information Technologies, vol. 787, pages 79–105, 2012. (Cited on pages 36, 37, 52 and 175.)

[Elsts 2013a] A. Elsts, J. Judvaitis and L. Selavo. *SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers.* In Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pages 220 – 227. EUROMICRO, IEEE, 2013. (Cited on pages 98 and 175.)

[Elsts 2013b] A. Elsts and L. Selavo. *Improving the Usability of Wireless Sensor Network Operating Systems.* In M. Ganzha, L. Maciaszek and M. Paprzycki, editeurs, Position Papers of the Federated Conference On Computer Science and Information Systems 2013 (FedCSIS 2013), pages 89–94. 2013. (Cited on pages 37 and 175.)

[Elsts 2014] A. Elsts, A. Mednis and L. Selavo. *Bayesian Network Approach to Vehicle Mode Monitoring Using Embedded System with 3-axis Accelerometer.* Special Issue on Artificial Intelligence in Robotics and Imaging of International Journal of Imaging & Robotics, vol. 12, no. 1, pages 67–80, 2014. ISSN 2231-525X. (Cited on pages 37, 92 and 176.)

[Faludi 2010] R. Faludi. Building wireless sensor networks: With zigbee, xbee, arduino, and processing. O'Reilly Media, Inc., 2010. ISBN 978-0596807733, 322 pages. (Cited on pages 20, 24 and 28.)

[Franceschinis 2009] M. Franceschinis, L. Gioanola, M. Messere, R. Tomasi, MA Spirito and P. Civera. *Wireless sensor networks for intelligent transportation systems.* In Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th, pages 1–5. IEEE, 2009. (Cited on page 160.)

[Ganesan 2002] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin and S. Wicker. *Complex behavior at scale: An experimental study of low-power wireless sensor networks.* Rapport technique, Technical Report UCLA/CSD-TR 02, 2002. (Cited on page 163.)

[García-Hernández 2007] Carlos F García-Hernández, Pablo H Ibarguengoytia-Gonzalez, Joaquín García-Hernández and Jesús A Pérez-Díaz. *Wireless sensor networks and applications: a survey.* IJCSNS International Journal of Computer Science and Network Security, vol. 7, no. 3, pages 264–273, 2007. (Cited on page 97.)

[Gay 2003] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer and D. Culler. *The nesC language: A holistic approach to networked embedded systems.* In ACM Sigplan Notices, volume 38, pages 1–11. ACM, 2003. (Cited on page 29.)

[Gay 2009] D. Gay, P. Levis, D. Culler and E. Brewer. *nesC 1.3 Language Reference Manual.* `http://nesl.ee.ucla.edu/fw/torres/home/projects/`

`tossim_gumstix/root/nesc-1.3.1/doc/ref.pdf`, 2009. [Online: accessed 08.05.2013.]. (Cited on pages viii, 76 and 131.)

[Green 1992] T. Green and M. Petre. *When visual programs are harder to read than textual programs.* In Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD. Citeseer, 1992. (Cited on pages 126 and 128.)

[Greenstein 2004] B. Greenstein, E. Kohler and D. Estrin. *A sensor network application construction kit (SNACK).* In Proceedings of the 2nd international conference on Embedded networked sensor systems, SenSys '04, pages 69–80, New York, NY, USA, 2004. ACM. (Cited on pages 31 and 128.)

[Gummadi 2005] R. Gummadi, O. Gnawali and R. Govindan. *Macro-programming Wireless Sensor Networks using Kairos.* Lecture Notes in Computer Science, Distributed Computing in Sensor Systems, pages 466–466, 2005. (Cited on pages 16, 29 and 34.)

[Halstead 1977] M. Halstead. Elements of software science (operating and programming systems series). Elsevier Science Inc., 1977. ISBN 978-0444002051, 142 pages. (Cited on page 133.)

[Handziski 2005] V. Handziski, J. Polastre, J. Hauer, C. Sharp, A. Wolisz and D. Culler. *Flexible Hardware Abstraction for Wireless Sensor Networks.* In Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005), 2005. (Cited on page 37.)

[Hill 2000] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister. *System architecture directions for networked sensors.* ACM Sigplan Notices, vol. 35, no. 11, pages 93–104, 2000. (Cited on pages 21 and 36.)

[Hudak 1997] P. Hudak. *Domain-specific languages.* Handbook of Programming Languages, vol. 3, pages 39–60, 1997. (Cited on page 98.)

[Jones 1994] C. Jones. *Software metrics: good, bad and missing.* Computer, vol. 27, no. 9, pages 98–100, 1994. (Cited on page 133.)

[Judvaitis 2012] J. Judvaitis. *Visual wireless sensor network programming environment.* Bachelor's thesis at University of Latvia, 2012. (Cited on page 177.)

[Judvaitis 2013] J. Judvaitis, A. Elsts and L. Selavo. *Demo Abstract: SEAL-Blockly: Sensor Network Visual Programming Using a Web Browser.* In Poster and Demo Proceedings of 10th European Conference on Wireless Sensor Networks (EWSN 2013), pages 71–74, 2013. (Cited on page 175.)

[Kaplan 2004] D.T. Kaplan. *Teaching computation to undergraduate scientists.* ACM SIGCSE Bulletin, vol. 36, no. 1, pages 358–362, 2004. (Cited on pages viii, 12 and 13.)

[Karl 2007] H. Karl and A. Willig. Protocols and architectures for wireless sensor networks. Wiley, June 2007. ISBN 9780470095102, 498 pages. (Cited on pages 10, 18, 19 and 122.)

[Klues 2009] K. Klues, C.J.M. Liang, J. Paek, R. Musăloiu-e, P. Levis, A. Terzis and R. Govindan. *TOSThreads: Thread-Safe and Non-invasive Preemption in TinyOS*. In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09), pages 127–140. ACM, 2009. (Cited on pages 21, 56 and 93.)

[Kothari 2007] N. Kothari, R. Gummadi, T. Millstein and R. Govindan. *Reliable and efficient programming abstractions for wireless sensor networks*. ACM SIGPLAN Notices, vol. 42, no. 6, pages 200–210, 2007. (Cited on pages 16, 30, 34, 97, 115, 129, 139, 140 and 150.)

[Krueger 1992] C. Krueger. *Software reuse*. ACM Computing Surveys (CSUR), vol. 24, no. 2, pages 131–183, 1992. (Cited on pages 7 and 17.)

[Lajara 2010] R. Lajara, J. Pelegrí-Sebastiá and J.J.P. Solano. *Power consumption analysis of operating systems for wireless sensor networks*. Sensors, vol. 10, no. 6, pages 5809–5826, 2010. (Cited on pages 19, 85 and 88.)

[Langendoen 2006] K. Langendoen, A. Baggio and O. Visser. *Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture*. In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pages 8–pp. IEEE, 2006. (Cited on pages 15, 16 and 171.)

[Lauer 1979] Hugh C Lauer and Roger M Needham. *On the duality of operating system structures*. ACM SIGOPS Operating Systems Review, vol. 13, no. 2, pages 3–19, 1979. (Cited on page 52.)

[Levis 2002] Philip Levis and David Culler. *Maté: A tiny virtual machine for sensor networks*. In ACM Sigplan Notices, volume 37, pages 85–95. ACM, 2002. (Cited on page 18.)

[Levis 2004a] P. Levis. *The TinyScript language: A Reference Manual*. `http://www.cs.berkeley.edu/~pal/mate-web/files/tinyscript-manual.pdf`, 2004. [Online: accessed 08.05.2013.]. (Cited on pages viii, 32, 127 and 131.)

[Levis 2004b] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler. *TinyOS: An operating system for sensor networks*. In Ambient Intelligence. Springer Verlag, 2004. (Cited on pages 18, 21 and 72.)

[Levis 2005] P. Levis, D. Gay, V. Handziski, J.H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk*et al.* *T2: A second generation os for* *embedded sensor networks*. Telecommunication Networks Group, Technische Universität Berlin, Technical Report TKN-05-007, 2005. (Cited on page 21.)

[Levis 2012] P. Levis. *Experiences from a Decade of TinyOS Development*. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), 2012. (Cited on pages 9, 20, 21, 36, 60 and 72.)

[Li 2003] S. Li, R. Sutton and J. Rabaey. *Low power operating system for heterogeneous wireless communication system*. In L. Benini, M. Kandemir and J. Ramanujam, editeurs, Compilers and operating systems for low power, pages 1–16. Springer, 2003. (Cited on page 19.)

[Lin 2012] F. Lin. *BlocklyDuino: a web-based visual programming editor for Arduino*. `https://github.com/gasolin/BlocklyDuino`, 2012. [Online: accessed 10.05.2013.]. (Cited on page 26.)

[LSI 2013] *Latvia State Institute of Fruit-Growing*. `http://www.lvai.lv`, 2013. [Online: accessed 25.02.2013.]. (Cited on page 158.)

[LU 2012] LU. *Latvijas Universitātes informācijas sistēma*. `http://luis.lv`, 2012. [Online: accessed 30.04.2012.]. (Cited on pages viii and 12.)

[Madden 2005] S.R. Madden, M.J. Franklin, J.M. Hellerstein and W. Hong. *TinyDB: an acquisitional query processing system for sensor networks*. ACM Trans. Database Syst., vol. 30, pages 122–173, March 2005. (Cited on pages 32, 34, 97 and 128.)

[Mainland 2008] G. Mainland, G. Morrisett and M. Welsh. *Flask: staged functional programming for sensor networks*. In Proceedings of the 13th ACM SIGPLAN international conference on Functional programming, ICFP '08, pages 335–346, New York, NY, USA, 2008. ACM. (Cited on pages 29 and 97.)

[Maloney 2008] J.H. Maloney, K. Peppler, Y. Kafai, M. Resnick and N. Rusk. *Programming by choice: Urban youth learning programming with Scratch*. ACM SIGCSE Bulletin, vol. 40, no. 1, pages 367–371, 2008. (Cited on pages 14, 97 and 126.)

[McCartney 2011] W.P. McCartney and N. Sridhar. *Stackless preemptive multithreading for TinyOS*. In Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on, pages 1–8. IEEE, 2011. (Cited on page 57.)

[Mednis 2011] A. Mednis, G. Strazdins, R. Zviedris, G. Kanonirs and L. Selavo. *Real time pothole detection using Android smartphones with accelerometers*. In Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on, pages 1–6. IEEE, 2011. (Cited on pages 92, 140 and 147.)

[Mednis 2012a] A. Mednis. *A Multimodal Approach for Determination of Vehicle Position.* In Workshops on Business Informatics Research, pages 223–235. Springer, 2012. (Cited on pages 140 and 149.)

[Mednis 2012b] A. Mednis, A. Elsts and L. Selavo. *Embedded solution for road condition monitoring using vehicular sensor networks.* In Application of Information and Communication Technologies (AICT), 2012 6th International Conference on, pages 1–5. IEEE, 2012. (Cited on pages 92 and 176.)

[Mednis 2012c] A. Mednis and R. Zviedris. *RFID Communication: How Well Protected Against Reverse Engineering?* In Proceedings of the 2nd International Conference on Digital Information Processing and Communications (ICDIPC12), pages 57–62. Klaipeda University, 2012. (Cited on page 140.)

[Miller 1956] George Miller. *The magical number seven, plus or minus two: Some limits on our capacity for processing information.* The psychological review, vol. 63, pages 81–97, 1956. (Cited on page 100.)

[Miller 2009] J.S. Miller, P.A. Dinda and R.P. Dick. *Evaluating a BASIC approach to sensor network node programming.* In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, pages 155–168. ACM, 2009. (Cited on pages 11, 30, 31, 32, 34, 100, 127, 131, 133, 153 and 155.)

[Mooney 1993] James D Mooney. *Issues in the specification and measurement of software portability.* In Proceedings of 15th International Conference on Software Engineering, Baltimore, 1993. (Cited on page 39.)

[Moteiv Corporation 2006] Moteiv Corporation. *Tmote Sky: Low Power Wireless Sensor Module.* `http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf`, 2006. (Cited on pages vi, 7 and 8.)

[Mottola 2006] L. Mottola and G.P. Picco. *Logical neighborhoods: A programming abstraction for wireless sensor networks.* Distributed Computing in Sensor Systems, pages 150–168, 2006. (Cited on pages 29, 97 and 129.)

[Mottola 2011] L. Mottola and G.P. Picco. *Programming wireless sensor networks: Fundamental concepts and state of the art.* ACM Computing Surveys (CSUR), vol. 43, no. 3, page 19, 2011. (Cited on pages 11, 25 and 97.)

[MSP 2011] *MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller (Rev. G).* `www.ti.com/lit/ds/symlink/msp430f1611.pdf`, 2011. [Online; accessed 13.08.2013.]. (Cited on pages 8 and 65.)

[Neal 1989] L.R. Neal. *A system for example-based programming.* In ACM SIGCHI Bulletin, volume 20, pages 63–68. ACM, 1989. (Cited on pages 14, 97 and 100.)

[Newton 2007] R. Newton, G. Morrisett and M. Welsh. *The Regiment macroprogramming system.* In Proceedings of the 6th international conference on Information processing in sensor networks, pages 489–498. ACM, 2007. (Cited on pages 30, 34 and 97.)

[Nielsen 1994] J. Nielsen. *Usability inspection methods.* In J. Nielsen and Robert L. Mack, editeurs, Usability inspection methods, chapitre Heuristic evaluation, pages 25–62. John Wiley & Sons, Inc., New York, NY, USA, 1994. (Cited on page 72.)

[Nielsen 2000] J. Nielsen. *Why You Only Need to Test with 5 Users.* `http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/`, 2000. [Online; accessed 15.05.2012.]. (Cited on pages vii and 152.)

[Ousterhout 1996] J. Ousterhout. *Why threads are a bad idea (for most purposes).* Presentation given at the 1996 Usenix Annual Technical Conference, 1996. (Cited on page 52.)

[Pane 1996] J. Pane and B. Myers. *Usability Issues in the Design of Novice Programming Systems,.* Human-Computer Interaction Institute Technical Report CMU-HCII-96-101, 1996. (Cited on pages 13, 17, 31, 97, 100, 101, 126, 132 and 155.)

[Picco 2010] G.P. Picco. *Software engineering and wireless sensor networks: happy marriage or consensual divorce?* In Proceedings of the FSE/SDP workshop on Future of software engineering research, pages 283–286. ACM, 2010. (Cited on pages 6, 15, 16 and 80.)

[ply 2012] *PLY (Python Lex-Yacc).* `http://www.dabeaz.com/ply/`, 2012. [Online: accessed 01.10.2012.]. (Cited on page 116.)

[Polastre 2005] J. Polastre, R. Szewczyk and D. Culler. *Telos: enabling ultra-low power wireless research.* In Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, pages 364–369. IEEE, 2005. (Cited on page 159.)

[pys 2013] *PySense: A language to program wireless sensor network at once.* `http://code.google.com/p/pysense/`, 2013. [Online; accessed 30.03.2013.]. (Cited on pages 30, 34 and 128.)

[Raza 2012] Usman Raza, Alessandro Camerra, Amy L Murphy, Themis Palpanas and Gian Pietro Picco. *What does model-driven data acquisition really achieve in wireless sensor networks?* In Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on, pages 85–94. IEEE, 2012. (Cited on page 138.)

[Reas 2003] C. Reas and B. Fry. *Processing: a learning environment for creating interactive Web graphics.* In ACM SIGGRAPH 2003 Web Graphics, pages 1–1. ACM, 2003. (Cited on page 28.)

[Robins 2003] A. Robins, J. Rountree and N. Rountree. *Learning and teaching programming: A review and discussion.* Computer Science Education, vol. 13, no. 2, pages 137–172, 2003. (Cited on pages 13, 14 and 97.)

[Romer 2004] K. Romer and F. Mattern. *The design space of wireless sensor networks.* Wireless Communications, IEEE, vol. 11, no. 6, pages 54–61, 2004. (Cited on pages 11, 47 and 97.)

[Rubio 2007] B. Rubio, M. Diaz and J.M. Troya. *Programming approaches and challenges for wireless sensor networks.* In Systems and Networks Communications, 2007. ICSNC 2007. Second International Conference on, pages 36–36. IEEE, 2007. (Cited on pages 6 and 25.)

[sea 2012] *The SEAL programming language.* `http://open-sci.net/wiki/seal`, 2012. [Online: accessed 14.05.2013.]. (Cited on page 116.)

[Selavo 2007] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young*et al.* *Luster: wireless sensor network for environmental research.* In Proceedings of the 5th international conference on Embedded networked sensor systems, pages 103–116. ACM, 2007. (Cited on pages 30, 139, 140 and 158.)

[Sensiron 2011] Sensiron. *SHT21.* `http://www.sensirion.com/en/pdf/product_information/Datasheet-humidity-sensor-SHT21.pdf`, 2011. [Online; accessed 30.03.2011.]. (Cited on page 122.)

[Seybold 2005] J.S. Seybold. Introduction to RF propagation. Wiley-Interscience, 2005. ISBN 978-0471655961, 352 pages. (Cited on pages 168 and 169.)

[Sharp 2007] C. Sharp, M. Turon and D. Gay. *TinyOS TEP 102: Timers.* `http://www.tinyos.net/tinyos-2.x/doc/html/tep102.html`, 2007. [Online: accessed 10.02.2013.]. (Cited on page 66.)

[Sieber 2013] A. Sieber and J. Nolte. *Online device-level energy accounting for wireless sensor nodes.* In EWSN'13 Proceedings of the 10th European conference on Wireless Sensor Networks, pages 149–164. Springer, 2013. (Cited on page 85.)

[Simon 2006] D. Simon, C. Cifuentes, D. Cleal, J. Daniels and D. White. *Java on the bare metal of wireless sensor devices: the squawk Java virtual machine.* In Proceedings of the 2nd international conference on Virtual execution environments, pages 78–88. ACM, 2006. (Cited on pages 32 and 34.)

[Sivieri 2012] A. Sivieri. *Erlang meets WSNs: a functional approach to WSN programming.* In Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on, pages 562–563. IEEE, 2012. (Cited on pages 28 and 34.)

[Smith 2007] Randall B. Smith. *SPOTWorld and the Sun SPOT.* In Proceedings of the 6th international conference on Information processing in sensor networks, IPSN '07, pages 565–566, New York, NY, USA, 2007. ACM. (Cited on pages 32 and 128.)

[Strazdins 2010] G. Strazdins, A. Elsts and L. Selavo. *MansOS: easy to use, portable and resource efficient operating system for networked embedded devices.* In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10, pages 427–428, New York, NY, USA, 2010. ACM. (Cited on page 175.)

[Strazdins 2011] G. Strazdins, A. Gordjusins, G. Kanonirs, V. Kurmis, A. Mednis, R. Zviedris and L. Selavo. *Team University of Latvia GCDC 2011 Technical Paper.* `http://www.gcdc.net`, 2011. (Cited on pages 140 and 146.)

[Strazdins 2013] Girts Strazdins, Atis Elsts, Krisjanis Nesenbergs and Leo Selavo. *Wireless Sensor Network Operating System Design Rules Based on Real-World Deployment Survey.* Journal of Sensor and Actuator Networks, vol. 2, no. 3, pages 509–556, 2013. (Cited on pages 7, 8, 27, 29, 42, 67, 97, 98, 99 and 175.)

[Suh 2006] Changsu Suh, Young-Bae Ko, Cheul-Hee Lee and Hyung-Joon Kim. *The Design and Implementation of Smart Sensor-based Home Networks.* Technical Report of R & D Departments, Hanback Electronics Company, Daejeon, Republic of Korea College of Information & Communication, Ajou University, Suwon, Republic of Korea, 2006. (Cited on page 160.)

[Tanenbaum 2006] Andrew S Tanenbaum and Albert S Woodhull. Operating systems: Design and implementation. Prentice Hall, 3rd édition, 2006. ISBN 0131429388, 1080 pages. (Cited on page 17.)

[Tavakoli 2007] A. Tavakoli, D. Chu, J.M. Hellerstein, P. Levis and S. Shenker. *A declarative sensornet architecture.* ACM SIGBED Review, vol. 4, no. 3, pages 55–60, 2007. (Cited on page 31.)

[Texas Instruments 2013] Texas Instruments. *MSP430 Ultra-Low Power 16-Bit Microcontrollers.* `http://www.ti.com/msp430`, 2013. [Online: accessed 04.04.2013.]. (Cited on page 8.)

[Titzer 2006] Ben L Titzer. *Virgil: Objects on the head of a pin.* In ACM SIGPLAN Notices, volume 41, pages 191–208. ACM, 2006. (Cited on page 128.)

[Tolle 2005] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay*et al. A macroscope in the redwoods.* In Proceedings of the 3rd international conference on Embedded networked sensor systems, pages 51–63. ACM, 2005. (Cited on page 158.)

[Van Deursen 1997] A. Van Deursen and P. Klint. *Little languages: little maintenance?* In SIGPLAN Workshop on Domain-Specific Languages. Citeseer, 1997. (Cited on pages 63 and 98.)

[Van Deursen 2000] A. Van Deursen, P. Klint and J. Visser. *Domain-specific languages: An annotated bibliography.* ACM Sigplan Notices, vol. 35, no. 6, pages 26–36, 2000. (Cited on pages 16 and 98.)

[Van Vliet 2008] H. Van Vliet. Software engineering: principles and practice, Third edition. Wiley, 2008. ISBN 978-0470031469, 740 pages. (Cited on pages 2 and 15.)

[Von Behren 2003] R. Von Behren, J. Condit and E. Brewer. *Why events are a bad idea (for high-concurrency servers).* In Proceedings of the 9th conference on Hot Topics in Operating Systems, volume 9, pages 4–4, 2003. (Cited on page 52.)

[Wang 2006] Qiang Wang, Yaoyao Zhu and Liang Cheng. *Reprogramming wireless sensor networks: challenges and approaches.* Network, IEEE, vol. 20, no. 3, pages 48–55, 2006. (Cited on page 18.)

[Wark 2007] T. Wark, P. Corke, P. Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain and G. Bishop-Hurley. *Transforming agriculture through pervasive wireless sensor networks.* IEEE Pervasive Computing, pages 50–57, 2007. (Cited on page 158.)

[was 2012] *Libelium WaspMote.* `http://www.libelium.com/products/waspmote`, 2012. [Online: accessed 06.10.2011.]. (Cited on pages 160 and 164.)

[Welsh 2004] M. Welsh and G. Mainland. *Programming Sensor Networks Using Abstract Regions.* In NSDI'04 Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. USENIX Association, 2004. (Cited on pages 6, 28 and 129.)

[Welsh 2010] Matt Welsh. *A Mote it is to Trouble the Mind's Eye: The Next Decade of Sensor Networking.* Keynote speech at the 7th European Conference on Wireless Sensor Networks (EWSN'10), 2010. (Cited on pages 1 and 166.)

[Werner-Allen 2006] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees and M. Welsh. *Fidelity and yield in a volcano monitoring sensor network.* In Proceedings of the 7th symposium on Operating systems design and implementation, pages 381–396. USENIX Association, 2006. (Cited on pages 29, 139, 140, 143 and 166.)

[Wheat 2011] D. Wheat. *Arduino Software.* In Arduino Internals, pages 89–97. Springer, 2011. ISBN 978-1430238829. (Cited on pages 20 and 24.)

[Wood 2006] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin and J. Stankovic. *ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring.* University of Virginia Computer Science Department Technical Report, 2006. (Cited on page 30.)

[Wood 2008] A. Wood, L. Selavo and J. Stankovic. *SenQ: An embedded query system for streaming data in heterogeneous interactive wireless sensor networks.* Distributed Computing in Sensor Systems, pages 531–543, 2008. (Cited on pages 30 and 34.)

[Zacepins 2010] A. Zacepins, J. Meitalovs and E. Stalidzans. *Model based real time automated temperature control system for risk minimization in honey bee wintering building.* In Proceedings of the 8th International Industrial Simulation Conference (ISC 2010), pages 245–247, 2010. (Cited on page 158.)

[Zolertia 2013] Zolertia. *Z1 Platform.* `http://www.zolertia.com/ti`, 2013. [Online: accessed 28.04.2013.]. (Cited on pages 8, 22 and 65.)

[Zviedris 2010] R. Zviedris, A. Elsts, G. Strazdins, A. Mednis and L. Selavo. *LynxNet: Wild Animal Monitoring Using Sensor Networks.* In P. Marron, T. Voigt, P. Corke and L. Mottola, editeurs, Real-World Wireless Sensor Networks, volume 6511 of *Lecture Notes in Computer Science*, pages 170–173. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17520-6_18. (Cited on pages 37, 91, 140, 145 and 175.)

Appendix A

SEAL grammar

⟨*program*⟩ ::= ⟨*declaration*⟩*

⟨*declaration*⟩ ::= ⟨*component_use_case*⟩
  | ⟨*when_block*⟩
  | ⟨*do_block*⟩
  | ⟨*system_configuration*⟩
  | ⟨*pattern_declaration*⟩
  | ⟨*const_statement*⟩
  | ⟨*set_statement*⟩
  | ⟨*define_statement*⟩
  | ⟨*load_statement*⟩
  | ';'

⟨*component_use_case*⟩ ::= 'use' ⟨*identifier*⟩ ⟨*parameter_list*⟩ ';'
  | 'read' ⟨*functional_expression*⟩ ⟨*parameter_list*⟩ ';'
  | 'output' ⟨*identifier*⟩ ⟨*output_fields*⟩ ⟨*parameter_list*⟩ ';'

⟨*system_configuration*⟩ ::= 'config' ⟨*identifier*⟩ ⟨*value*⟩ ';'

⟨*pattern_declaration*⟩ ::= 'pattern' ⟨*identifier*⟩ '(' ⟨*value_list*⟩ ')' ';'

⟨*const_statement*⟩ ::= 'const' ⟨*identifier*⟩ ⟨*value*⟩ ';'

⟨*set_statement*⟩ ::= 'set' ⟨*identifier*⟩ ⟨*functional_expression*⟩ ';'

⟨*define_statement*⟩ ::= 'define' ⟨*identifier*⟩ ⟨*functional_expression*⟩
  ⟨*parameter_list*⟩ ';'

⟨*load_statement*⟩ ::= 'load' ⟨*string_literal*⟩ ';'

⟨*functional_expression*⟩ ::= ⟨*identifier*⟩ '(' ⟨*argument_list*⟩ ')'
  | ⟨*identifier*⟩ ⟨*value*⟩
  | ⟨*value*⟩

⟨*argument_list*⟩ ::= ⟨*functional_expression*⟩ ',' ⟨*argument_list*⟩
  | ⟨*functional_expression*⟩

⟨*output_fields*⟩ ::= '(' ⟨*output_field_list*⟩ ')'
  | ε

⟨*output_field_list*⟩ ::= ⟨*output_field_spec*⟩ ’,’ ⟨*output_field_list*⟩
  | ⟨*output_field_spec*⟩

⟨*output_field_spec*⟩ ::= ⟨*identifier*⟩
  | ⟨*identifier*⟩ ⟨*integer_literal*⟩

⟨*when_block*⟩ ::= ’when’ ⟨*condition*⟩ ’:’ ⟨*declaration*⟩* ⟨*elsewhen_block*⟩ ’end’

⟨*elsewhen_block*⟩ ::= ’elsewhen’ ⟨*condition*⟩ ’:’ ⟨*declaration*⟩* ⟨*elsewhen_block*⟩
  | ’else’ ’:’ ⟨*declaration*⟩ *
  | ε

⟨*do_block*⟩ ::= ’do’ ⟨*parameter_list*⟩ ’:’ ⟨*declaration*⟩* ⟨*then_block*⟩ ’end’

⟨*then_block*⟩ ::= ’then’ ⟨*parameter_list*⟩ ’:’ ⟨*declaration*⟩* ⟨*then_block*⟩
  | ε

⟨*condition*⟩ ::= ⟨*condition_term*⟩
  | ⟨*condition*⟩ ’or’ ⟨*condition_term*⟩

⟨*condition_term*⟩ ::= ⟨*condition_factor*⟩
  | ⟨*condition_term*⟩ ’and’ ⟨*condition_factor*⟩

⟨*condition_factor*⟩ ::= ⟨*logical_statement*⟩
  | ’not’ ⟨*condition_factor*⟩

⟨*logical_statement*⟩ ::= ⟨*functional_expression*⟩
  | ’(’ ⟨*condition*⟩ ’)’
  | ⟨*functional_expression*⟩ ⟨*equality_comparison*⟩ ⟨*functional_expression*⟩
  | ⟨*functional_expression*⟩ ⟨*nonequality_comparison*⟩ ⟨*functional_expression*⟩
  | ⟨*functional_expression*⟩ ’>’ ⟨*functional_expression*⟩
  | ⟨*functional_expression*⟩ ’⟨’ <*functional_expression*⟩
  | ⟨*functional_expression*⟩ ’>=’ ⟨*functional_expression*⟩
  | ⟨*functional_expression*⟩ ’⟨=’ <*functional_expression*⟩

⟨*parameter_list*⟩ ::= ⟨*parameter_list*⟩ ’,’ ⟨*parameter*⟩ | ε

⟨*parameter*⟩ ::= ⟨*identifier*⟩
  | ⟨*identifier*⟩ ⟨*value*⟩
  | ’pattern’ ⟨*value*⟩
  | ’parameters’ ⟨*value*⟩
  | ’where’ ⟨*value*⟩

⟨*value_list*⟩ ::= ⟨*value_list*⟩ ’,’ ⟨*value*⟩ | ⟨*value*⟩

⟨*value*⟩ ::= ⟨*boolean_literal*⟩
  | ⟨*floating_point_literal*⟩
  | ⟨*integer_literal*⟩

| ⟨*string_literal*⟩
| ⟨*complex_identifier*⟩

⟨*complex_identifier*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩ ’.’ ⟨*identifier*⟩

⟨*equality_comparison*⟩ ::= ’==’
| ’=’

⟨*nonequality_comparison*⟩ ::= ’!=’
| ’<>’

⟨*boolean_literal*⟩ ::= ’true’
| ’false’

⟨*integer_literal*⟩ ::= ⟨*decimal_literal*⟩
| ⟨*hexadecimal_literal*⟩

⟨*decimal_literal*⟩ ::= digit + alphabetic character *

⟨*hexadecimal_literal*⟩ ::= ’0x’ hex_digit +

⟨*floating_point_literal*⟩ ::= ⟨*decimal_literal*⟩ . digit +

⟨*string_literal*⟩ ::= ’ ” ’ (any character except quote and newline) * ’ ” ’

⟨*identifier*⟩ ::= alphabetic character (alphabetic character | digit) *

All syntax is case-insensitive.