Improving the Usability of Wireless Sensor Network Operating Systems

Atis Elsts*† and Leo Selavo*†
*Faculty of Computer Science
University of Latvia, Riga, Latvia
†Institute of Electronics and Computer Science
Riga, Latvia
Email:{atis.elsts,selavo}@edi.lv

***Abstract*—Wireless sensor network research community has constructed a number of operating systems that enable development of sensor network applications using novel and appropriate software abstractions. Unfortunately, the abstractions are not always easily usable by inexperienced users, because the learning curves of these existing operating systems are quite steep.**

In this article we present selected usability aspects of *MansOS* wireless sensor network operating system. We believe that MansOS has the potential to make sensor network programming accessible for a broader range of programmers. The evaluation of MansOS suggests improved usability for non-expert users over TinyOS and other operating systems without compromising efficiency.

I. Introduction

ISO standards [1] 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 paper, 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.

The need for better WSN software usability has long been recognized in the research community. A wireless sensor network operating system should provide clean and flexible services that allow the developer to create highly-concurrent low duty-cycle applications naturally.

TinyOS [2] was the first to address this challenge. It is small, efficient and has been highly influential. Unfortunately, the system has gained notoriety as being difficult to learn [3]. Sensor network operating systems developed after TinyOS tried to increase usability and learnability in several ways: by allowing to write applications in plain C [4], using preemptive multithreading [5], or supporting familiar, UNIX-like abstractions [6].

We believe that while TinyOS did great by introducing many new abstractions suitable for WSN, more work could have been devoted making these abstractions accessible for a broad 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, including those without WSN experience.

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

- Abstractions that are either already familiar or easy to grasp (Section III-C, Section III-D).
- Interchargeable components that can be enabled or disabled without changing application source code (Section III-B).
- Improved portability (Section III-A).
- Reduced source code size and prototyping time for simple sense-and-send applications (Section IV-A, Section IV-B).

The hard challenge was to implement usability while not giving up on WSN-specific design goals, in particular, efficiency.

II. Related Work

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 [7].

TinyOS is written (and requires applications to be written) in a custom 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 (for example, static virtualization, parametrized components), the fact that concurrency is fully exposed to the user, and high generality and modularity of the OS itself [3].

In contrast to many early attempts to apply TinyOS to real-world deployments, in more recent years TinyOS powered sensor networks often turn out to be a complete success. Nevertheless, finding new contributors to the core code is challenging [3]. The authors of TinyOS admit that over-generalization and too finely grained components add to the difficulty of 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.

Contiki is a lightweight operating system created at the Swedish Institute of Computer Science [4]. Contiki applications are written in plain C, support dynamic loading and unloading of components, and run on top of many popular WSN hardware platforms (including TelosB).

Contiki is recognized for its unique (in WSN context) execution model: on top of event-based kernel lightweight cooperative threading primitives are implemented, called *protothreads*

[8]. Although not without limitations, protothreads is a simple and elegant alternative both to event-driven application code and to preemptive multithreading. Most of stateful system services (networking etc.) are implemented as protothreads in Contiki.

MANTIS OS is a WSN OS created at University of Colorado at Boulder [5].

Multithreading is a key design feature of Mantis OS, added to increase the usability of the OS. Multithreading allows 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.

Mantis is designed to work on multiple hardware platforms. In particular, application code can also be compiled to run natively on x86 architecture.

LiteOS [6] 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. By reusing concepts from UNIX operating system, LiteOS tries to reduce the steepness of the learning curve, because the existing knowledge of the system programmer is leveraged. However, the suitability of these concepts to WSN is not always clearly shown.

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

Other operating systems and software libraries like **Arduino** [9] can be used to build sensor networks, but they are not WSN-specific and not well-optimized for the task.

III. MansOS Aspects and Abstractions

A. Code Organization

At the moment, MansOS supports ten hardware platforms, but only three hardware (MCU) architectures: Texas Instruments MSP430, Atmel AVR, and x86. Therefore, a distinction is made between platform-specific and architecture-specific code. This simple invention allows to greatly reduce the proportion of platform specific code.

MansOS code organization has both similarities and differences to TinyOS and Contiki. One remarkable difference is the size of *platform*-specific code, which is much lower in MansOS (because *architecture*-specific code is introduced). 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 the periodic timer interrupt handler code (the "heartbeat" of the system) is unified and shared by all platforms.

For example, the official Contiki code for XM-1000 sensor device provided by AdvancticSYS [10] 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 XM-1000 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.

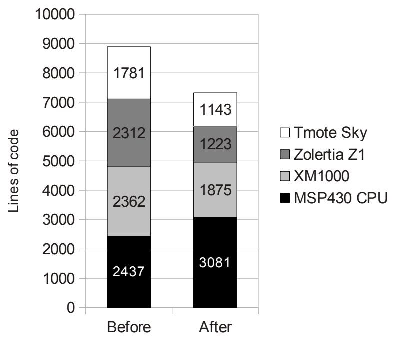


Fig. 1: The result of applying MansOS code organization to three Contiki hardware platforms and MSP430 MCU architecture

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. 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 neccessary to prove the point of this optimization. The result (excluding platform-specific example application code) is shown in Fig. 1. 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.

B. The Component Selection Mechanism

TinyOS allows the application developer to specify which components to use in application's 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, to extensibility and flexibility, as it now becomes simple to integrate platform-specific or application-specific components in the OS.

In contrast, C-based operating systems like Contiki and Mantis have have limited means to achieve something similar. Contiki adopts "everything is included by default" strategy, which leads to large code size (Fig. 4). 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.

The solution that MansOS adopts is to use an additional file next to application's source code: a *configuration file*. The configuration file allows to select or exclude components to use in the specific application. The resulting component granularity is larger than in TinyOS, but this approach is also useful for setting system-wide policies, e.g. the radio channel 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.

MansOS build system ensures that each platform provides a reasonable set of components that are enabled by default. The dependencies between components are also resolved automatically. The novice user may successfully start using MansOS without being aware that the component selection mechanism exists. On the other hand, the experienced user may optimize or extend his applications by excluding and including components manually.

There are three benefits brought by such a mechanism:

- Modularity: components that implement the same interface can be used interchangeably, often without changing application C source code. For example, two versions of the kernel (event-based and thread-based) can be replaced with each other by changing just one line in 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 single line in application's configuration file.
- Efficiency: unused components are not included in the binary image. This allows to achieve more efficient binary code and RAM usage, which in turn reduces prototyping and debugging time (Section IV-B).

The MansOS component selection mechanisms allows to achieve both function-granularity and file-granularity garbage collection at the link stage. The latter is needed to exluded components that are referenced by the kernel at initializion, but are not used neither by application code nor other components.

In order to show that MansOS approach effectively scales to other sensor network operating systems, we partially implemented the component selection algorithm for Contiki as well. The implementation 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. In contrast to the project-specific configuration file option already present in Contiki, this new mechanism is much more comprehensive, generic, allows to exclude whole files from build, and is enabled by default.

C. 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.

The cause of the problem lies in the fact that hardware timer ticks used for time accounting do not have high-accuracy mapping to milliseconds. The high-precision (20 ppm error), but low-frequency crystal has "binary", 32 768 Hz frequency.

Mantis is triggering the interrupt in *approximately* the needed time interval and adding the timer ticks passed since last interrupt to a system time counter. Unfortunately, the timing error accumulates with time.

TinyOS puts the responsibility for time correction on the user. The accounted milliseconds are called "binary milliseconds" in TinyOS; every 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 [11] is so generic and multifaceted that introducing such a fix in it is much harder.

The Contiki solution is to not to provide millisecond timer abstraction 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.

Timing accuracy is important in WSN, as we discovered ourselves in a precision agriculture deployment [12]. The users of the system wanted to record timestamped microclimate data from multiple locations. Although time-synchronization protocols for WSN exist (even with *micro*second precision, for example, [13]), and MansOS includes a simple version of such a protocol, it was of limited help in this situation: the network often become partitioned because some of intermediate nodes died. The motes we were using had neither a Real Time Clock chip nor a GNSS receiver, therefore software-only solution was required.

Millisecond abstraction improves usability, because is allows to use familiar rather than unfamiliar measurement unit.

[1]https://www.millennium.berkeley.edu/pipermail/tinyos-help/

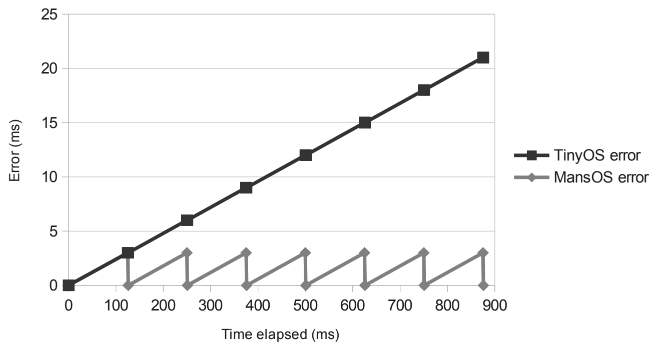


Fig. 2: Timing accuracy comparison

The idea adopted in MansOS is to use two hardware timers: a "counter" timer running with *approximately* 1000 Hz frequency, and a "correction" timer running with *precisely* 8 Hz frequency. Every time the counter timer fires, the global millisecond time counter is increased by one. Every time the "correction" fires, the counter is decreased by three. The accuracy error, which is cumulative in TinyOS, is kept bounded to 3 milliseconds in MansOS (Fig. 2).

Losing short-time accuracy is the trade-off of MansOS solution. 3 millisecond error is not noticeable by a human, but might make a difference when, for example, time slots for high-contention TDMA MAC protocol need to be allocated with high precision. In such a case, the user is better off by using hardware timers directly.

D. Memory allocation

Initially the memory for system-level objects (sockets, files, queued packets) in MansOS was allocated statically, using array variables at file-scope. 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.

A 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. This pattern is used for opening both files and sockets. 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 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.

Levis [3] praises parametrized components implemented in TinyOS as one of its most worthy features. However, we show that nesC is not required to implement an abstraction similar to parametrized hardware components. 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. Such an application requires that the USART module is initialized in serial protocol mode. TinyOS generates code specifically for USART #1; 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, one for each USART, and a high-level, inline wrapper function *serialInit()*. If GCC linker optimizations are enabled, the resulting code is as small as in TinyOS, because the USART #0 function code is optimized away.

IV. Evaluation

In this section, MansOS is experimentally evaluated and compared to TinyOS, Contiki, and Mantis (latest publicly available versions on 18th May, 2013). LiteOS is not included in the comparison as it lacks TelosB support.

Four test applications were analyzed: *Active* – busy looping application, *RadioRx* – radio packet reception, *RadioTx* – radio packet transmission, *Combined* – senses internal voltage and temperature, sends values to radio and writes them to the external flash memory[2].

A. Source Code Measurements

In this test, application source code length is compared. Lines of code are counted 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[3].

C code is approximately at the same level of abstraction as nesC code. Therefore, by comparing the size of the source code, approximate information about application complexity written in these languages can be deduced.

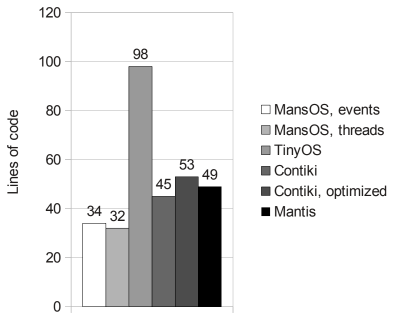


Fig. 3: Source code size comparison of the *combined* application

The results are provided in Fig. 3. Lower source code size in MansOS suggests better usability compared to other OS, especially to TinyOS. (The "optimized" version of Contiki required additional lines in application configuration file.)

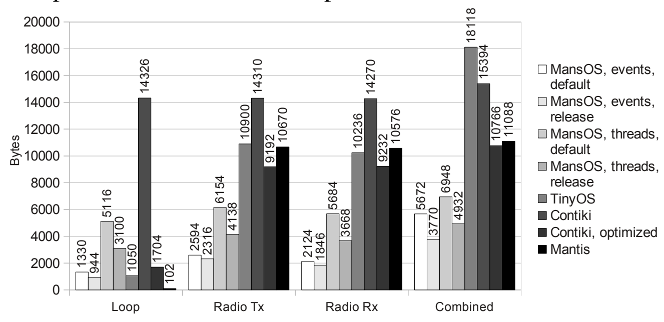
[2]Implementations available at http://mansos.edi.lv/dissert/testapps.tgz
[3]Available at http://mansos.edi.lv/wiki

B. Binary Code Measurements

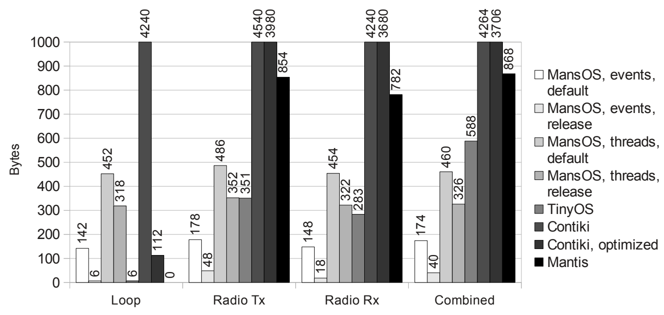
The code sizes were obtained by compiling with msp430-gcc version 4.5.3, using `-Os` optimization level. MansOS and Contiki builds also enabled linker optimizations, achieving garbage collection of unused functions.

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. Certain parts of debugging code (e.g. the *ASSERT* macro) remained even in the "release" versions.

Two versions of each application were also developed for Contiki. The first set was used 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. In order to show the improvements possible in this default configuration mechanism, a second, "optimized" set of application versions was developed on top of the optimized version of Contiki that partially incorporated the MansOS component selection mechanism.



(a) Flash memory usage



(b) RAM usage

Fig. 4: Application flash memory and RAM usage comparison

The results are given in Fig. 4 (Contiki RAM usage not shown in full). We note that the universality of the comparison is limited, as more advanced algorithms are used in some operating systems. For example, the external flash driver in TinyOS automatically adds CRC to data records, while in MansOS and other operating systems raw memory access takes place. However, the goal of this comparison is not as much to determine which OS can implement the same functionality in shorter code, but rather to determine the usability for out-of-the-box OS, including how big typical application prototypes are. Therefore the default coding style (platform-independent wherever possible) and build options are used.

There are various reasons why the other operating systems use more resources than MansOS. TinyOS uses resources for hardware component virtualization. Contiki includes most of functionality available on the platform by default. Mantis includes a lot of debugging code, for example, descriptive ASCII strings and code that operates LEDs. However, the core functionality of the system, for example the scheduler, also is implemented with larger overhead than in MansOS [14].

The optimized version of Contiki demonstrates good results for this set of applications: at least 30.0 % reduction in flash usage and at least 12.3 % reduction in RAM usage, showing that the MansOS component selection mechanism is a viable way how to reduce superfluous resource usage in other operating systems as well.

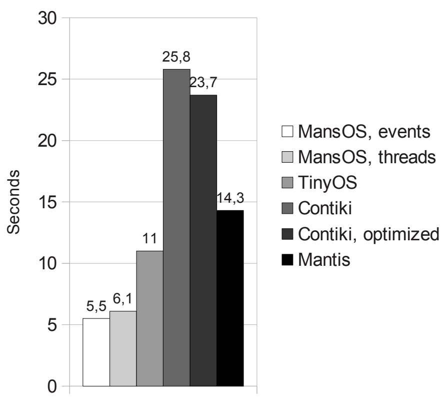


Fig. 5: Time of compilation and upload of the *combined* application

Binary code size (at least indirectly) has significant impact on developer productivity. If 20 second shorter binary code upload takes place, for a network of ten nodes the difference is 3 min 20 sec in total, but for 100 nodes: more than half an hour. Even though not many novice developers are going to work with 100-node networks, many of them are going to create and upload a lot of versions of their software while trying to debug a problem. Shorter code sizes make this process much faster (Fig. 5).

C. Execution Flow Analysis

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, called repeatedly by the system alarms processor.

In MansOS multithreaded version, 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 posted and ready to run. If no task is posted, the scheduler puts system in a low power mode. Each interrupt in turn causes the associated event handler to run.

The MansOS code usually is smaller not because it is missing a 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.

V. Conclusions

We have presented aspects of MansOS and compared its usability with several existing wireless sensor network operating systems.

For non-expert users, MansOS improves usability over other C-based WSN OS by implementing semi-automatic high-level component selection mechanism that allows to easily extend, modularize, and optimize applications and the OS itself. Compared to the other WSN OS analyzed, it is the only one that offers *accurate* millisecond-precision software timers and time accounting on TelosB platform.

Some of the results (component selection mechanism and four-layer code architecture) have been adapted to Contiki with good results (more than 10 % improvements in resource usage and source code size, respectively).

MansOS adopts several features from TinyOS, for example, static memory allocation, extensive compile-time and link-time optimizations to remove unused code, and some aspects of parametrized hardware interfaces. It improves the usability of these features by implementing them in plain C.

Acknowledgements

We want to thank to all who have contributed to MansOS in these years since 2008. Girts Strazdins is thanked for his help in the user study, and Andrey Vihrov for valuable MansOS improvement ideas.

References

[1] A. Abran, A. Khelifi, W. Suryn, and A. Seffah, "Usability meanings and interpretations in ISO standards," *Software Quality Journal*, vol. 11, no. 4, pp. 325–338, 2003.

[2] 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, pp. 93–104, 2000.

[3] 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.

[4] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 455–462.

[5] 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, pp. 563–579, 2005.

[6] 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*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 233–244.

[7] 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.

[8] 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*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42.

[9] J. Brock, R. Bruce, and S. Reiser, "Using Arduino for introductory programming courses," *Journal of Computing Sciences in Colleges*, vol. 25, no. 2, pp. 129–130, 2009.

[10] AdvanticSYS, "XM1000," http://www.advanticsys.com/wiki/index.php?title=XM1000, 2013, [Online: accessed 26.04.2013.].

[11] 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.].

[12] 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)*, 2012, pp. 271–281.

[13] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 39–49.

[14] A. Elsts, G. Strazdins, A. Vihrov, and L. Selavo, "Design and Implementation of MansOS: a Wireless Sensor Network Operating System," *Computer Science and Information Technologies, Scientific Papers University of Latvia*, vol. 787, pp. 79–105, 2012.