Improving the Usability of Wireless Sensor Network Operating Systems

Atis Elsts and Leo Selavo
Faculty of Computing, University of Latvia,
19 Raina Blvd., Riga, LV 1586, Latvia
and
Institute of Electronics and Computer Science,
14 Dzerbenes Str, Riga, LV 1006, Latvia
{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 broader range of programmers. The evaluation of MansOS suggests improved usability for non-expert users over TinyOS and other operating systems without compromising efficiency.

Keywords: Sensor networks, operating systems

1. Introduction

ISO standards [AKSS03] 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 low duty-cycle applications naturally, and handle highly concurrent execution flow reliably.

TinyOS [HSW+00] 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 [Lev12]. Sensor network operating systems developed after TinyOS tried to increase usability in several ways: by allowing to write applications in plain C [DGV04], using preemptive multithreading [BCD+05], or supporting UNIX-like abstractions [CASH08].

As the sensor network research field matures, there is an increasing need for an operating system targeted towards applications rather than research itself. MansOS [ESVS12] is a WSN OS developed with this goal in mind. It is implemented in plain C; the objective of MansOS since inception has been declared as being 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.

This publication is an extended version of a position paper with the same title [ES13] presented at Federated Conference on Computer Science and Information Systems, 2013.
2. 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 [LMP+04].

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 [Lev12].

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 [Lev12]. 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 [DGV04]. 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 [DSVA06]. 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 [BCD+05].
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 [CASH08] 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 [BBR09] can be used to build sensor networks, but they are not WSN-specific and not well-optimized for the task.

3. Code architecture

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 MantisOS to new hardware platforms, the code architecture is designed to reduce platform-specific code proportion. The design ideas author has developed include: separate platform-specific
Improving WSN OS Usability

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.

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. Architecture-specific code roughly corresponds to the 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 (kernel/main.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 4). The platform-specific components 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 in some cases (when the access functions cannot be inlined for some reason). However, high data bandwidth support is usually not among the primary requirements of WSN applications [SENS13].

For a case study of a specific hardware platform we are going to take AdvantecSYS XM1000 sensor device [Adv13]. The official Contiki code (evaluated including Makefiles, excluding comments and empty lines) for XM1000 sensor device provided by AdvantecSYS 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 1](image.png)

**Figure 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 AdvancicSYS 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 nec-
necessary 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. 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.

4. 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). 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.
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 circular dependencies between the libraries which cannot be resolved by the GCC linker.
Improving WSN OS Usability

Figure 2 – In the application development and configuration process, user code is added and a subset of MansOS components is selected and configured.
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 be 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. 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 to enable preprocessing of this header. Therefore usability is decreased, especially for inexperienced users.

The syntax of a MansOS configuration file is very simple; it consists of variable assignments (Listing 1). 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.

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 con-

configuration file allows to write nothing more than variable assignments, but nesC syntax includes more details.

Listing 1 – Example MansOS configuration file

```
1 #
2 # Configuration file of a multihop routing testing application
3 #
4 # Use threaded kernel
5 USE_THREADS=y
6 # Use nondefault radio channel
7 USE_RADIO_CHANNEL=11
8 # Use SAD routing and MAC protocols
9 # Use roles as data forwarder in the network (other roles are possible)
10 USE_ROLE_FORWARDER=y
11 # Enable debugging
12 DEBUG=y
```

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.
Application source code

System source code: C files

Device-independent source code
C files

Architecture-specific source code
C files

Platform-specific source code
C files

System source code: C files

User creates

C file(s)

Configuration file

List of config. options

List of compilation flags

Are generated

Are filtered

Included C files

Are compiled

Object files

Are pruned

Included object files

Are linked

Executable

Default configuration

Configuration file

Architecture-specific configuration

Platform-specific configuration

System source code: configuration files

Note: the join process allows configuration options to be redefined (overridden)

Note: component-granularity pruning takes place.
This action is described by Algorithm 2

Note: function-granularity pruning also takes place here

Figure 3 – 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). This allows to achieve more efficient binary code and RAM usage, which in turn reduces prototyping and debugging time (Section 6.1).

The whole build process is shown in Fig. 3 and elaborated in more technical detail in Algorithm 1.

The pseudo-code of Algorithm 2 is also given (it corresponds to the “Are pruned” step in Fig. 3). 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.
Algorithm 1 The component selection algorithm

Include the default configuration file
Include platform-specific configuration file
Include application-specific configuration file, if present

▷ Overrides default settings
▷ 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 ← application's .c files`
`systemSourceFiles ← hardware-independent $\cup$ architecture-$\cup$ platform-specific .c files`
`enabledSourceFiles ← \emptyset`

for `file ∈ allSourceFiles` do
    `component ← the component this file belongs to`
    if `USE_component = y` then
        `enabledSourceFiles ← enabledSourceFiles \cup \{file\}`
    end if
end for

▷ Compile all potentially required files

`applicationObjectFiles ← COMPIL\{e\}(applicationSourceFiles)`
`systemObjectFiles ← COMPIL\{e\}(systemSourceFiles)`

▷ Link the required files together

`executable ← LINK(FIND\{R\}EQUIRED\{F\}ILES(applicationObjectFiles, systemObjectFiles))`
▷ The function FIND\{R\}EQUIRED\{F\}ILES is defined in Algorithm 2
Algorithm 2 Pruning of unused components

function \texttt{FINDREQUIREDFILES}(\texttt{appFiles}, \texttt{systemFiles})

\begin{itemize}
  \item librarySymbols $\leftarrow \{\text{`printf', `malloc', \ldots}\}$ \Comment{All functions from C library}
  \item symbolsFound $\leftarrow$ librarySymbols
  \item symbolsUnresolved $\leftarrow \emptyset$ \Comment{Determine which symbols are required by the application}
\end{itemize}

\For{file $\in$ \texttt{appFiles}}

\begin{itemize}
  \item localExports $\leftarrow$ symbols exported from file
  \item localImports $\leftarrow$ external symbols required to file
  \item symbolsUnresolved $\leftarrow$ symbolsUnresolved $\cup$ (localImports \symbol{\texttt{\textbackslash}} symbolsFound)
  \item symbolsUnresolved $\leftarrow$ symbolsUnresolved $\setminus$ localExports
  \item symbolsFound $\leftarrow$ symbolsFound $\cup$ localExports
\end{itemize}

\EndFor

\begin{itemize}
  \item result $\leftarrow$ \texttt{appFiles} \Comment{Determine which files are required; start with all application’s files}
  \item \texttt{systemFiles} $\leftarrow$ \texttt{systemFiles} \Comment{Process system files while all symbols are found or all files included}
\end{itemize}

\texttt{progress} $\leftarrow$ \texttt{true}

\While{symbolsUnresolved $\neq \emptyset$ AND \texttt{systemFiles} $\neq \emptyset$ AND \texttt{progress}}

\begin{itemize}
  \item \texttt{progress} $\leftarrow$ \texttt{false}
  \item \For{file $\in$ \texttt{systemFiles}}

  \begin{itemize}
    \item localExports $\leftarrow$ symbols exported from file
    \item localImports $\leftarrow$ external symbols required to file \Comment{Does this file include new symbols?}
  \end{itemize}

  \If{symbolsUnresolved $\setminus$ localExports $= \emptyset$} \texttt{continue} \EndIf

  \item symbolsUnresolved $\leftarrow$ symbolsUnresolved $\setminus$ localExports
  \item symbolsUnresolved $\leftarrow$ symbolsUnresolved $\setminus$ (localImports \symbol{\texttt{\textbackslash}} symbolsFound)
  \item symbolsFound $\leftarrow$ symbolsFound $\cup$ localExports
  \item result $\leftarrow$ result $\cup$ \{file\} \Comment{This file will be linked}
  \item \texttt{progress} $\leftarrow$ \texttt{true}
\EndFor
\EndWhile

\Return result
\EndFunction
The best results are achieved when both file-granularity and function-granularity garbage collection are combined (Section 6.1, Fig. 8). Now the question may arise – why there is a need for file-granularity binary code component selection (Algorithm 2) when the finer-grained function-granularity selection is already in place? 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 we know that the component is not used, it is safe to exclude the component from the binary image. Algorithm 2 does exactly that: excludes components that have these properties. GCC extension __attribute__((weak)) is used to signal that the GCC linker may leave the reference to the component’s initialization function unresolved in the final image.

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 1, and does not include Algorithm 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\(^1\). The results are promising and are discussed in the evaluation section of this chapter (Section 6.1).

5. 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-

\(^1\) The modified version of the Contiki operating system is available at https://github.com/atiselists/contiki-optimized.
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 allow to have 1:1 mapping to milliseconds in whole numbers. In TelosB-based and similar platforms (Tmote Sky, Zolertia Z1 [Zol13], AdvanticSYS XM1000 [Adv13], 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 [MSP11]). 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 WSN motes 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.

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 wall-clock second contains 1024 binary milliseconds. For novice users, this approach is confusing, as nothing in the name of Tmi11i 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 list1.

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 [STG07] 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.

Timing accuracy is important in WSN, as we discovered ourselves in a precision agriculture deployment [EBJS12]. The users of the system wanted to record timestamped microclimate data from multiple locations. Although time-synchronization protocols for WSN exist (even with microsecond precision, for example, [MKSL04]), 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.

![Timing accuracy comparison](image)

Figure 4 – Timing accuracy comparison

The idea adopted in MansOS is to use two hardware timers (Algorithm 3): 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. 4).

Algorithm 3 The essential elements of MansOS time accounting algorithm

```latex
function MSP430INITCLOCKS \triangleright Called during system initialization
milliseconds \leftarrow 0
wire TimerA to the low-frequency crystal \triangleright 32 768 ticks per second
set TimerA0 period to 32 ticks \triangleright 1024 times per second
set TimerA1 period to 4096 ticks \triangleright 8 times per second
start TimerA
end function

function TIMER0INTERRUPT \triangleright Called on TimerA0 hardware interrupt
milliseconds \leftarrow milliseconds + 1
end function

function TIMER1INTERRUPT \triangleright Called on TimerA1 hardware interrupt
milliseconds \leftarrow milliseconds - 3
end function
```

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.

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 the most recent revisions available on 18th May, 2013.

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.\(^1\)

\(^1\) The implementations of these applications are available at http://mansos.edi.lv/dissert/testapps.tgz
6.1. Binary code size and RAM usage

The RAM usage and code sizes were obtained by compiling with \textit{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 \texttt{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 \texttt{ASSERT} macro) remained even in the “release” versions.

![Figure 5 – Application binary size comparison](image)

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. \texttt{nullmac\_driver}, \texttt{nullrdoc\_driver}, and \texttt{framer\_nullmac} options were selected for all of the applications, and \texttt{CONTIKI\_NO\_NET} define was also selected in Makefile of the \textit{loop} application. Then, in order to show the improve-
All of the WSN OS analyzed try to reduce binary code size in some way. MansOS: by using the configuration mechanism (Algorithm 1 and Algorithm 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 sorting 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 4), and by more aggressive inlining because of higher-granularity inline functions. The effects caused by disabling these techniques are show in Fig. 8 (for MansOS; this, in contrast to the rest of results in this chapter, was evaluated on MansOS revision 981) and Fig. 9 (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 component-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 [CASH08] 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 monolithic binary image that includes the whole kernel functionality.

Figure 7 – Application compile & upload time comparison of the combined application
Figure 8 – Effects of enabling link-level optimizations in MansOS (event-based, “release” version)

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. 9): at least 30.1% 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.
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 `nos_load_display()` function in kernel code. The problems are caused by circular dependencies of the libraries. We can conclude that increasing the number of separately inked 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 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. 7). 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 [Pic10]. 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. 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 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.

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. 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 combined application 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).

6.2. 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 runtime 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.

6.3. 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-bound.

We compared MansOS with TinyOS and Contiki. 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.

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 re-
quires explicit calls to `sleep()` function. Although this usage pattern
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 manage-
ment 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 1 and in Fig. 10. 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 en-
ergy 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). 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 millisece-
Table 1 – Average, minimal and maximal current consumption of the combined application. In brackets: standard deviation of the three samples, expressed as % of value

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

onds for TinyOS; however, the slope at the sides of this peak is not steep, and increased energy consumption (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 10 – Average, minimal and maximal current consumption of the combined application (logarithmic scale)](image)

**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.

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 [LPSS10] that Contiki consumes more energy than other popular WSN operating systems. This is primarily because the system by default configures and turns on most of the functionality 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 μA 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 μA even in the sleep mode.

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 extension 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 high 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 conclusion section of this paper.) Therefore, it makes sense to compare energy efficiency of just this core part, rather than of the whole networking stack.

7. 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.

Compared to the state of art operating systems already existing for years, MansOS often lacks advanced functionality. This is an area of future work. However, clear separation must be made between missing functionality that can be added without fundamental changes in OS ar-
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. In contrast, an example of the latter kind of work is a fully automatic mechanism of resource arbitration, but it is not present in other C-based WSN operating systems as well.

Acknowledgements

We want to thank to all who have contributed to MansOS in these years since 2008.

References


Improving WSN OS Usability


