SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2012. Vol. 787
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 79–105 P.

Design and Implementation of MansOS: a Wireless Sensor Network Operating System

Atis Elsts[1,2], Girts Strazdins[1,2], Andrey Vihrov[1,2], and Leo Selavo[1,2]

[1] Faculty of Computing, University of Latvia,
19 Raina Blvd., Riga, LV-1586, Latvia
[2] Institute of Electronics and Computer Science,
14 Dzerbenes Str., Riga, LV-1006, Latvia
atis.elsts@lu.lv, girts.strazdins@lu.lv,
andrejs.vihrovs@edi.lv, leo.selavo@lu.lv

Abstract. Wireless sensor networks (WSN) consist of a number of low-power spatially distributed autonomous nodes equipped with means of communication. Developing software for WSN is challenging; operating system (OS) support is required. In this paper we present MansOS, a portable and easy-to-use WSN operating system that has a smooth learning curve for users with C and UNIX programming experience. The OS features a configuration model that allows to reduce application binary code size and build time. In contrast to other WSN OS, MansOS by default provides both event-based and threaded user application support, including a complete although lightweight implementation of preemptive threads.

Keywords: Sensor networks, operating systems.

1 Introduction

Few areas of embedded system programming require OS support more than WSN programming does. Networks formed by autonomous small-scale devices with tight resource and energy constraints are systems of great complexity. Developing fully application-specific solutions without using middleware or OS is not an option for majority of users. MansOS is an open-source operating system designed to serve their needs.

WSN programming is challenging because it brings together complexity of embedded device programming and complexity of networked device programming. Therefore an easy-to-use OS with a smooth learning curve is needed. Since a lot of system programmers have experience with C programming and UNIX-like concepts, it makes sense to adapt these concepts to WSN programming. MansOS is written in plain C, aims to be user-friendly and use familiar concepts.

In contrast to their desktop counterparts, embedded hardware architectures and platforms come in great variety and are often specially adapted to concrete applications. Therefore portability is a critical requirement for embedded software. MansOS is portable and runs on several WSN mote platforms.

The design and implementation of a feature complete WSN OS is not a quick or easy task. In some form, MansOS has been under development since 2007. It started off as a LiteOS clone: an attempt to bring portability to this WSN OS. During the development, MansOS has been influenced by ideas from Contiki and Mantis.

Existing WSN OS in some cases use unnecessarily heavy technologies and suboptimal implementations. For example, OS support for threads can be simplified and optimized by taking into account traits specific to WSN OS software: the low number of total threads expected and cooperativeness of user threads. Moreover, code of existing WSN OS is often bloated by forcing the use of unnecessary resources and components. MansOS brings these simplifications and optimizations to WSN OS area, and allows smaller resource use granularity.

The paper starts with an overview of related work. Then a section is devoted to description of MansOS architecture, and hardware design space. It then proceeds with a description and evaluation of selected components, and concludes with a comparison of MansOS and competing solutions.

2 Related work

Multiple operating systems for sensor networks have been proposed previously, focusing on different design aspects. Notable examples include TinyOS, Contiki, LiteOS and Mantis.

2.1 TinyOS

TinyOS [11] is an actively supported and well tested WSN OS, has created a wide contributor and user community, and can be considered *de facto* standard for WSN programming. It is primarily targeted to sensor network researchers.

Compact, reactive scheduler is used (core system uses 400 bytes of program memory). Source code is written in *nesC* language, a C dialect processed by a *nesC* parser and pre-compiled into a single C source file. This single file is then compiled into a firmware image and takes advantages of static compiler optimizations.

TinyOS is a highly modular system, consisting of components, wired together using specified interfaces. Each component provides a particular service and interfaces describe *commands* for starting a service and *events* for signalling completion of a service routine. Inside components low-priority tasks are scheduled, using non-preemptive, run-to-completion FIFO task queue. High-priority event handlers are used for time-critical section execution. Optional preemptive scheduler can be used, implemented as an add-on, called TOSThreads [10].

Although TinyOS is portable, it has a steep learning curve for novice WSN developers. The main reasons of TinyOS complexity are:

- event-driven nature of TinyOS. Event-based code flow is more complex for programmers to design and understand, as state machine for split-phase operation of the application has to be kept in mind;

- nesC language concepts, unfamiliar even to experienced C and embedded system programmers.

These limitations are in the system design level, and there is no quick fix available. The most convenient alternative is to implement middleware on top of TinyOS for simplified access to non-expert WSN programmers.

2.2 Contiki

Contiki is a lightweight operating system with support for dynamic loading and replacement of individual programs and services. It is built around an event-driven kernel and provides optional preemptive multithreading [6]. Contiki is written in C language and has been ported to a number of platforms, including TelosB and Zolertia Z1, having different CPU architectures: Atmel AVR, Texas Instruments MSP430 and others.

The only abstractions provided by Contiki kernel are CPU multiplexing and dynamic program and service loading. Additional abstractions are provided by libraries with full access to underlying hardware. Loadable programs are implemented, using modified binary format containing relocation information and performing run-time relocation.

Contiki provides proto-threads abstraction: ability to write thread-like programs with blocking calls on top of event-driven kernel [7]. Each proto-thread requires only two bytes of memory, and has no separate stack. As an alternative to cooperative proto-threads, Contiki provides preemptive threading model implemented as optional library. It uses separate stacks for each thread and context switching, which must be implemented for every CPU, if used. However, to the best of our knownledge, there are no platforms whose implementations of threads support preemption.

Process is either an application or a service – a process implementing functionality used by multiple applications. Both applications and services may be replaced at run-time. Communication between services is implemented using event passing through kernel.

The porting of Contiki consists of writing platform-specific boot up code, device drivers, architecture specific dynamic program loading and context switching for preemptive multithreading. The kernel and service layer are platform-independent.

Contiki programs are relatively heavy-weight, as they usually use several kilobytes of RAM and have program code more than 10 kilobytes in size. Core routines of the system are sometimes duplicated between platforms, for example, timer interrupt handlers are custom for every MCU architecture. Architecture-independent scheduler of preemptive threads is a desired feature that is not included in the OS by default.

2.3 LiteOS

LiteOS is a multithreaded operating system that provides Unix-like abstractions for wireless sensor networks [4]. It offers hierarchical file system, remote shell,

dynamic application loading, preemptive scheduler for multithreaded applications and and object oriented programming language LiteC++ – a subset of C++. LiteOS has been implemented on MicaZ and Iris mote platforms, both with AVR microcontrollers.

LiteOS utilizes a specific binary image format containing relative addresses to provide dynamic application loading.

LiteOS includes a hierarchical, UNIX-like file system, called LiteFS, for storage of application sensor data and application binary images. All files, including device drivers, provide unified POSIX-compatible access functions: `fopen()`, `fclose()`, `fseek()`, `fread()`, `fwrite()`, etc. Therefore also all sensor node devices are included in the file system accessible from remote shell.

Split phase operations, such as sending a radio message and waiting for end of transmission, are implemented as blocking calls. External events, such as reception of a radio message or external pin interrupt, were implemented as callback function handlers in the first versions of LiteOS, but were transformed to blocking calls later. Therefore LiteOS provides fully threaded programming with blocking calls, and no event callback handling.

To access kernel functions from user threads, system calls or *call gates* are used. Therefore function implementations can be changed at run-time, and kernel image updated without modification of user application.

The source code is 8-bit AVR platform-specific and significant changes are required to port LiteOS to other platforms with other microcontrollers.

2.4 Mantis

Mantis is a multithreaded cross-platform embedded operating system for wireless sensor networks, supporting complex tasks such as compression, aggregation and signal processing, implemented in a lightweight RAM footprint that fits in less than 500 bytes of memory, including kernel, scheduler, and network stack [3]. Mantis is implemented on multiple platforms, including PCs and PDAs, allowing to create hybrid networks consisting of real sensor nodes and virtual ones, simulated on top of one or multiple PCs. Written in C language, Mantis OS translates to a separate API on the PC platform, which can be augmented by any other required functionality – graphical user interface, data bases, web services.

Mantis provides preemptive, time-sliced thread scheduling and synchronization using semaphores. Each thread occupies a separate stack. Each context switch is performed by the kernel task scheduler in timer interrupt handler, and uses only approximately 120 instructions. All other interrupts besides timer are handled by device drivers directly.

Additional abstraction layer is implemented in the kernel, providing blocking call interface for external event waiting (radio or UART packet reception). Data buffer pool is used to share buffers between all communication layer services.

Mantis kernel code is platform-independent. However, platform- and chip-level code is mixed, there are no TelosB or MicaZ platforms, only MSP430 and AVR code, which is microcontroller (MCU) or architecture specific. Separation of MCU architectures, specific chips and platforms would improve portability.

3 MansOS architecture

In the first part of this section the MansOS hardware abstraction model is specified. After that, the design space of a portable WSN OS is outlined, by describing the specific architectures and hardware modules we chose to support.

3.1 Abstraction model

As one of our design goals is minimization of effort required to port the OS to new WSN hardware platforms, MansOS provides modular architecture. 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 frequently is usable across multiple platforms.

Hardware abstraction model in MansOS (Fig. 1) is based on a key observation from [9]: 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 the user has access to all four hardware abstraction 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 Arduino, TelosB or Zolertia Z1).
- platform independent code, including the *hardware interface layer (HIL)*, directory `hil`.

The HIL code provides unified device interface for kernel and user applications. Wiring, function binding and platform or architecture-specific constants are defined at `arch` and `platform` levels. To take an example, radio driver's interface is defined in the HIL level. During compilation time, the interface is bound to a specific implementation, which is chosen at the `platform` level, containing the glue code. For TelosB platform, CC2420 radio driver is chosen, and so on.

The model explicated here is similar to the one found in Contiki: `platforms` directory in MansOS roughly corresponds to `platforms` directory, `arch` to `cpu` in Contiki, `chips` to `core/dev`, and the rest of MansOS system (`kernel`, `hil`, and `lib`) to the rest of `core` folder in Contiki. The chief difference between these systems is better organization of chip- and platform-specific code in MansOS; for example, the periodic timer interrupt handler code (the "heartbeat" of the system) is unified and shared by all platforms. Another difference is function binding: in MansOS it is done earlier, at compile time. This design decision allows reducing both binary code size and RAM usage, as well as run-time overhead. To take a concrete example, in Contiki the radio driver is accessed through function

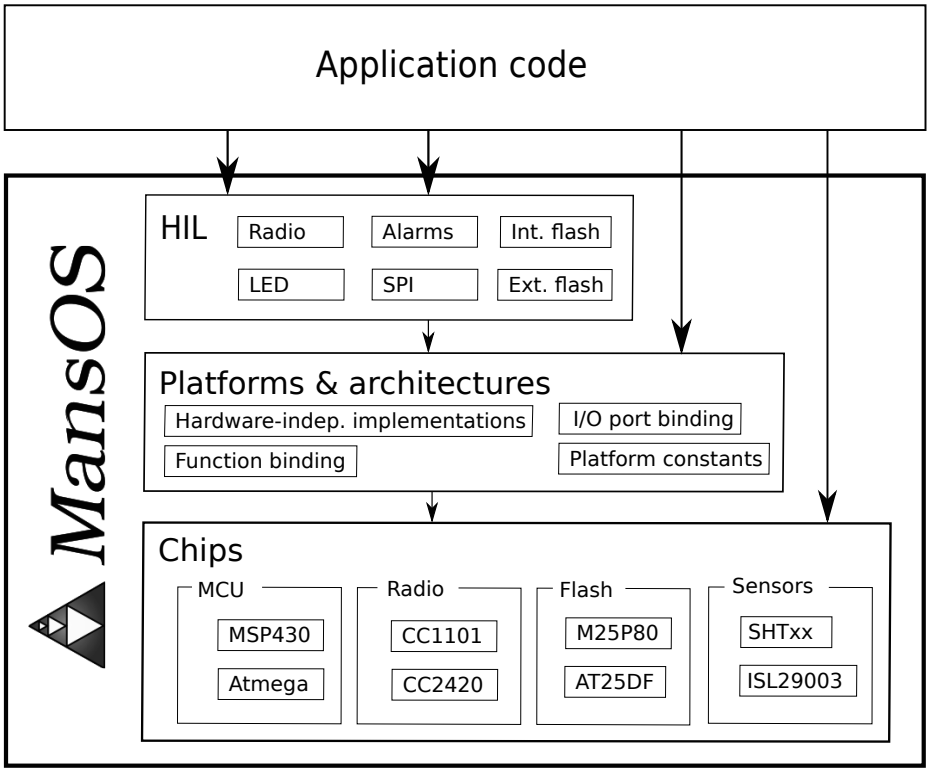


Fig. 1: MansOS components and abstraction layers

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 (glued by inline functions or macros), therefore extra resources are not used.

Similar parallels can be drawn between MansOS and TinyOS, although the latter lacks explicit separation of architecture-specific code: `platforms` in MansOS maps to `platforms` in TinyOS, `chips` to `chips`, `hil` to `interfaces`, `kernel` to `system`, `lib` to `lib`. A notable difference is the impossibility of direct hardware component access in TinyOS application code. It could be argued that this restriction leads to a better code organization, but we feel that it is too limiting to the user.

As the analysis shows (Table 1), approximately half of total MansOS code is hardware-independent. Since the amount of hardware-dependent code varies greatly with the number of hardware platforms supported, comparison is more fair when a specific platform is fixed. When TelosB is selected as the platform,

Table 1: Source code size breakdown with regard to MansOS components, lines of code (excluding comments and empty lines)

	All platforms		**TelosB**	
Chip-specific code	7132	34.61%	2657	19.86%
Architecture-specific code	2063	10.01%	582	4.35%
Platform-specific code	1482	7.19%	208	1.56%
Interface layer code	1814	8.8%	1814	13.56%
Kernel code	1240	6.02%	1240	9.27%
Network protocol code	3683	17.87%	3683	27.53%
File system code	1384	6.72%	1384	10.35%
Library code	1809	8.78%	1809	13.52%
Total device-independent code	9930	48.19%	9930	74.23%
Total code	**20607**	**100%**	**13377**	**100%**

only a quarter of the code turns out to be chip- or platform-specific. Most of the hardware-dependent code is plain C; ASM is used only in a few, specific places, such as thread context switching.

3.2 Hardware support

A wireless sensor network consists of a number of distributed devices that are not only small and low-cost, but also have to be powered from relatively low capacity batteries. Therefore computing power limitations are extremely tight. Typical WSN mote resources include ten to a few hundreds kilobytes of program memory, a few kilobytes of RAM, no memory protection or mapping support. The microcontroller usually has a few MIPS of computing power and features several lower energy consumption modes. A WSN OS has to make good use of the limited resources available. Energy efficiency is of paramount importance: the OS has to provide options for low duty cycling.

Among typical architectures used in for WSN sensor motes and resource constrained embedded devices, Texas Instruments MSP430 and Atmel AVR are prominent.

MSP430 [14] is a 16-bit, low-power microcontroller using MIPS architecture. The device features a number of peripherals useful for a WSN device. Digital and analog I/O pins are provided, as are multiple hardware timers, including pulse-width modulation (PWM), and watchdog. Analog inputs can be sampled using the built-in ADC circuitry, while digital pins allow specific data transfer protocols to be used, for example, U(S)ART, SPI, and I^2C access. Notable platform examples are TelosB-compatible motes, such as Tmote Sky [13], as well as newer developments like Zolertia Z1 [16].

Atmel AVR [2] is an 8-bit modified Harvard architecture RISC microcontroller. Integrated ADC, watchdog and multiple timers with PWM are present as well, as are digital and analog I/O ports. One difference between the two

architectures is related to flash memory access: MSP430 use unified memory address space for RAM and flash, while in AVR macros have to be used for program memory access. Equivalently powerful AVR chips use more energy than MSP430, therefore they are better suited to application domains where energy requirements are less stringent, such as automotive applications or building automation. Notable platform examples include Arduino [1] and Waspmote [12], both using *Atmega* series MCU.

Several peripherals (sensors and actuators) are usually present on the mote. The OS therefore has to provide support for digital data transfer bus protocols (UART, SPI, I^2C) typically used for communication with the peripherals. At least, API to the MCU built-in hardware support has to be provided. However, hardware-only support is not sufficient for all times, as our experience shows. For example, several slightly different I^2C protocol versions are used on different peripheral devices (such as light sensors). The hardware support for I^2C on MSP430 fails to take into account these differences. Therefore, a configurable software implementation of the protocol has to be provided by the WSN OS in order to properly communicate with these devices. Finally, platform-independent API for the most popular sensors (voltage, light, humidity and temperature) can be expected.

Time accounting is an essential feature of the WSN OS, since WSN users often require sensor measurements to be timestamped. As real time clock (RTC) chips are seldom present on WSN motes, the OS has to emulate one using MCU hardware timers.

Finally, support for at least wireless communication has to be included in the OS, since it is by far the most popular form of communication used in sensor networks. A frequently encountered design option is IEEE 802.15.4 compatible transceiver chips using 2.4 GHz frequency band. Support of such a chip can be expected from the WSN OS. Support for IEEE 802.15.4 MAC layer is optional, as WSN applications typically use WSN-specific MAC protocols.

4 MansOS components and features

This section describes selected MansOS components in detail, namely the configuration mechanism, kernel, threads, file system, and the reprogramming mechanism. The section is concluded with a technical discussion about usability and portability. Although interesting, the description of MansOS network stack goes beyond the scope of this paper. This custom stack has support for network addressing, MAC protocols, multi-hop routing, and pseudo-sockets. IPv6 support is available as an external third-party library by using uIPv6 [8].

4.1 Configuration mechanism

Many users are worried that using a WSN OS as opposed to writing all code in application-specific way leads to bloated code sizes and inefficient resource usage. MansOS configuration mechanism is designed to deal with these problems. As

non-intrusiveness is one of our design goals, MansOS provides reduced program sizes and seamless OS integration with application code. In fact, no MansOS services are required to be used – the OS can function as a simple library of frequently used routines.

MansOS configuration mechanism is a key feature of the system, underlying whole component selection and implementation. The mechanism is based on observation that the need for run-time re-configuration or WSN applications is small. In contrast to desktop systems, where threads and processes are created and die constantly, on small resource-constrained systems resource allocation is usually static. Consequently, the allocation can be done at compile time.

The benefits of rich compile-time configuration include:

- code size reduction by explicitly selecting used and unused components;
- more flexible resource usage by providing custom, more compact versions of the code for most frequenctly used scenarios, and for cases when resources are severely constrained. For example, a simplified scheduler is used in the default case of only two threads;
- application code complexity reduction, because run-time reconfiguration support in applications becomes less important. Run-time adaptation to resource allocation is often not necessary, since the compile-time system is flexible enough;
- run-time overhead reduction by compile-time binding. This allows both reducing processing overhead, since direct function calls are cheaper than calls by pointer, and reducing RAM & flash usage overhead, since there is no need to store device driver structures for indirect access.

The objective of the configuration mechanism is to achieve the modularity and heavy optimizations made possible by using *nesC* in TinyOS, but without the complexity of having to learn a new programming language. Therefore, the challenge is to emulate specific features of component-oriented programming using plain C and GNU make.

The interactive part of the configuration mechanism is implemented using configuration files. The files are hierachical: a system-wide default configuration template is used as the base, to which platform-specific, site-local, and application-specific changes are added. Relations between components are possible: there can be either a dependence relation (A requires B) or conflict relation (A cannot be used together with B).

The non-interactive part is implemented using GCC and GNU binutils support. The optimization has two independent stages. First, after the compilation process all object files are sorted in two sets: the set reachable (via function calls) from user code and the set unreachable from user code. Only the reachable files are passed to the linker. The second stage is based on a linker feature which allows to discard unused code sections. The method can be used only when each function has been put in separate code section by the compiler, but provides finer-grained optimization if active.

4.2 Kernel

In an embedded operating system, the two main functions of OS kernel are the initialization of the system and execution of the main loop. The kernel should be as small and non-intrusive as possible and feature energy saving support.

Initialization MansOS components are initialized in the `main()` function. First, for platform-specific initialization, `initPlatform()` routine is called. This routine is custom for each platform. Second, generic component initialization is done, as the latter can depend on the former. The next action taken after all initialization is completed, depends on programming model used. For event-based execution, `appMain()` is called. For threaded execution, two threads (user and kernel) are created and OS scheduler started.

Alternatively, by specifying a configuration option, the user can completely disable kernel code. The only requirement in that case: all components used should be properly initialized from user code.

Execution models Two application execution models are used in WSN OS:

- event-based (asynchronous);
- thread-based (synchronous).

Event-based model is simpler and requires less resources: scheduler code is not included in the OS, and thread stacks do not use extra RAM. On the other hand, this model is more challenging for the programmer, especially for one who is developing lenghty applications. For event based execution, program flow is not reflected in the source code. In this way event-based programming is similar to using `goto` operator, as in both cases the user has to keep in mind a complicated mental model of program's states.

The benefits of thread-based model can be observed in application code, as it becomes easier to write and understand. On the other hand, this approach is not only more heavyweight, but application execution becomes more difficult to trace, stack overflow errors as well as race conditions become possible, and the OS kernel becomes more challenging to implement correctly. Taking all this into account, MansOS offers both models and lets the user choose.

Event-based execution This 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 100 times per second (user-configurable value). Therefore, timers with precision up to 10 ms are available by default.

Similar callbacks can be registered for packet reception, whether serial or radio. User callbacks are executed immediately after hardware signals arrival of

new data, therefore delay is the smallest 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 all of OS code has to be reentrant.

Energy efficiency in this model can be achieved by calling one of `sleep()` family functions in application's main loop.

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. In most of cases, as single user thread is sufficient, if blocking function calls are allowed in it. Second, in contrast to desktop OS, threads in WSN OS can be expected to be cooperative. The first observations motivates the OS to provide simpler scheduler version by default, supporting only two threads. The second allows to forget about time-slicing and similar fairness guarantees.

Correct locking is a big issue in multithreaded software architectures. If the locking is not correct, race-conditions can lead to corrupt data, or deadlocks can occur. Even if the locking is correct, significant code size overhead still remains. The locking in a WSN OS kernel can be simplified by making the kernel thread to run with higher priority. MansOS thread implementation is hierarchical: user threads are one hierarchy level below the kernel thread. The kernel thread is used for system event processing only and cannot be interrupted by user threads, while user threads can interrupt each another.

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 all threads that are ready to run).

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

Listing 1 Thread stack guard

```
#define STACK_GUARD() do {                                          \
        /* declare a stack pointer variable */                      \
        MemoryAddress_t currentSp;                                  \
        /* read the current stack pointer into the variable */      \
        GET_SP(currentSp);                                          \
        /* compare the current stack pointer with stack bottom, */ \
        /* and abort in case of overflow */                         \
        ASSERT_NOSTACK(currentSp >= STACK_BOTTOM());                \
    } while (0)
```

Stack overflow is a nasty and hard-to-detect problem when threads with small and constant-sized stacks are used. To alleviate the detection of this problem, MansOS includes *stack guards* (Listing 1) – code fragments that can be put in functions most likely to be in the bottom of the call chain. The guard immediately aborts program execution in case an overflow is detected.

Energy efficiency using threaded execution can be achieved by calling one of `sleep()` functions in the main loops of every user thread. The system will enter low power mode if no threads (including the kernel thread) are active.

4.3 File system

A typical task for a WSN node is data logging for later relaying and analysis, since immediate transmission is not possible in all cases. Most WSN nodes include a flash chip for this purpose. However, using these chips directly by low-level device commands is non-trivial. Often it is needed to distinguish amongst several logical data streams and dynamically allocate space between them, as well as deal with the chips' hardware limitations. A WSN operating system should therefore provide a clean and easy interface to the data storage and deal with the hardware details.

MansOS features a simple file system that abstracts the physical storage as a number of logical files or streams. Following the MansOS philosophy, the file system interface is synchronous (UNIX-like) and thread-safe. In addition to basic file commands, the system has non-buffering and integrity-checking modes. On the low level, the system is designed for flash chips that have very large segments and don't contain integrated controllers that handle data rewrites and wear levelling.

A flash memory segment is the minimal unit of memory cells that can be erased at once (flash memory cells need to be erased before repeated writes). Segments can be several hundred kilobytes big depending on the flash type and model.

Data organization The file system divides physical storage – flash memory – in *data blocks* of fixed size. A file is a linked list of data blocks; new blocks are allocated on demand. Contrary to the contiguous storage approach used by some WSN file systems (Coffee [15]), this allows for dynamic file sizes at no cost. The next block's number is stored at the end of the current one.

The size of a data block is chosen so that there is low overhead from traversing and allocating blocks, yet so that there isn't much space loss from incomplete blocks. One flash segment contains a small number of data blocks, so that there is smaller chance for multiple files to occupy one segment. On the TelosB platform, which has a flash with 64 KB big segments, they are divided in four 16 KB data blocks, giving the total of 64 data blocks chip-wide.

For integrity checking, data blocks are further divided into *data chunks*, which fit into the WSN node's memory and have a checksum appended. This allows to detect errors without reading the data twice. The overall division of flash memory into smaller elements is shown in Fig. 2.

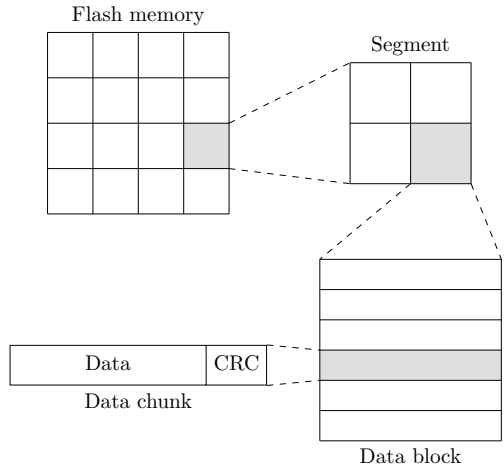


Fig. 2: Structural elements in the flash memory

Flash memory limitations on rewriting individual cells make the naïve approach of updating a file's contents in-place impractically slow. Some implementations use log-structured file system approach to solve this (the ELF file system [5]). But, since sensor data are sequential, the benefit of data rewrites may not justify the complexities they incur. Following the "keep it simple" principle, the MansOS file system disallows data rewrites completely; data can only be appended to a file.

Data block management Information about data blocks is held in the *block table*, a bitmap containing the current state of each data block (Fig. 3). The block table is small enough to be stored in the WSN node's EEPROM memory, where it can also be easily updated.

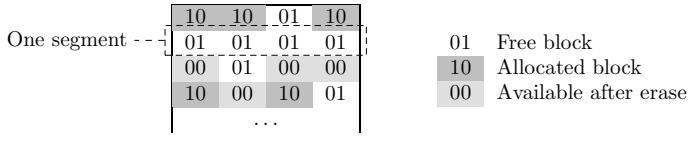


Fig. 3: Block table format

A data block can be in one of the three states: 1. free; 2. allocated; 3. available after erase (the erase operation needs to be performed on the block before it is usable again). After first time initialization, all blocks are in state 3.

The data block allocation procedure searches for usable blocks in the block table and assigns them to files on demand. It also attempts to decrease the number of blocks in state 3 that share a segment with another file (and cannot

be readily used) and to equalize the number of erases each data block is subjected to, thus performing flash memory wear levelling.

For this, free data blocks are probed in the following sequence, until one is found:

1. free blocks in the same segment as the previous data block of the file in question;
2. blocks in an empty segment;
3. blocks in an empty segment that needs to be erased before use;
4. other free blocks.

At each step, the block to allocate is chosen randomly from all available.

In the worst case, for steps 2–4 the procedure has to look at all data blocks in all segments. This can be improved by bit-packing data block states in one segment into one machine word and using bitmasks to determine the overall state of each segment.

Control structures File entries are kept in the root directory, which is also stored in the EEPROM memory. A file entry contains file name, first block number, file size and other fields. To keep the code size smaller, there is no support for hierarchical directory structure.

In-memory, open files are represented by two-tier structures, where a common part contains a file entry cache, reference count and a synchronization mutex, while the per-thread parts store file positions and read/write buffers. The use of buffers allows the flash chip to be in low-power mode most of the time.

4.4 Run-time management and reprogramming

Management 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.

In order to perform run-time management an efficient protocol (named *SSMP*) is implemented. Each available resource on the mote is assigned an object ID, which uniquely identifies the resource in mote-local scope. 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.

A command line shell is implemented to provide user access to the management interface (Fig. 4). The shell can be used both interactively and non-interactively from scripts. The shell allows to access arbitrary object IDs, although shortcuts for the most frequently used 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 motes in the network are accessed; in the second, only a single mote specified by its address is queried.

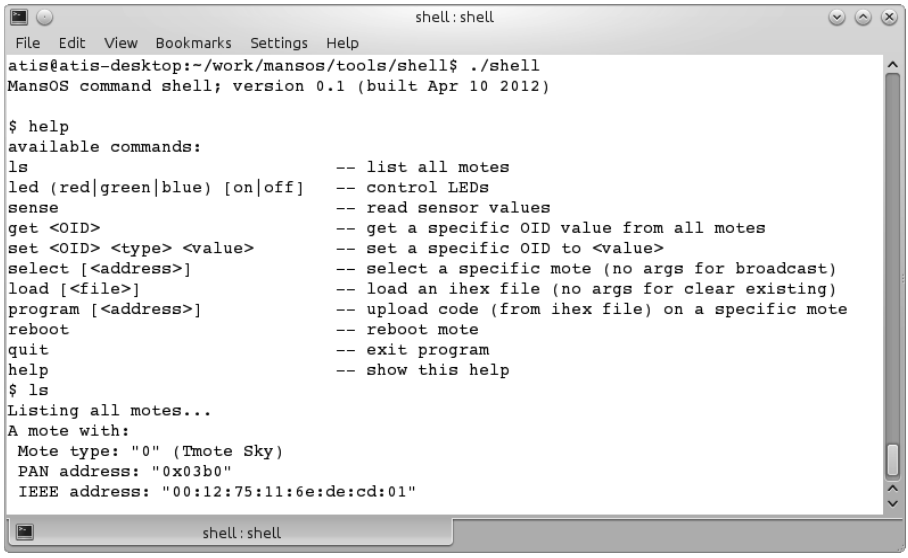


Fig. 4: MansOS shell

Reprogramming The need for run-time reprogramming support in WSN OS is apparent. For example, in our environmental monitoring use case, reprogramming just nine motes in outdoor conditions took more than two hours. Using over-the-air reprogramming reduces time requirements by an order of magnitude.

Run-time reprogramming in MansOS is performed in four stages:

1. binary code is read from file on disk and sent out to the WSN;
2. the code is transported through the network;
3. the code is received and stored on the target motes;
4. target motes reprogram themselves and run the received code.

The *first stage* is performed by MansOS shell (Fig. 4), to which a base station's mote is attached. For this purpose, the shell is extended with Intel IHEX file parser and multiple reprogramming related commands. For the *second stage*, SSMP is reused. A special object ID signalling binary data is used for code packets. The *third stage* is done by the reprogramming component of MansOS, which is included in application's binary image by specifying `USE_REPROGRAMMING=y` in its configuration file.

MansOS bootloader is responsible for the *final stage*, the most complicated one. 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, before reboot, the reprogramming component has signalled the need to replace the existing program image, the bootloader performs the actual rewrite: it replaces MCU program memory contents with a new OS image taken from mote's external storage. For safety, use of "golden image" is supported. If bootloader detects that the system has failed to start multiple times in a row, it loads last usable OS image from the storage.

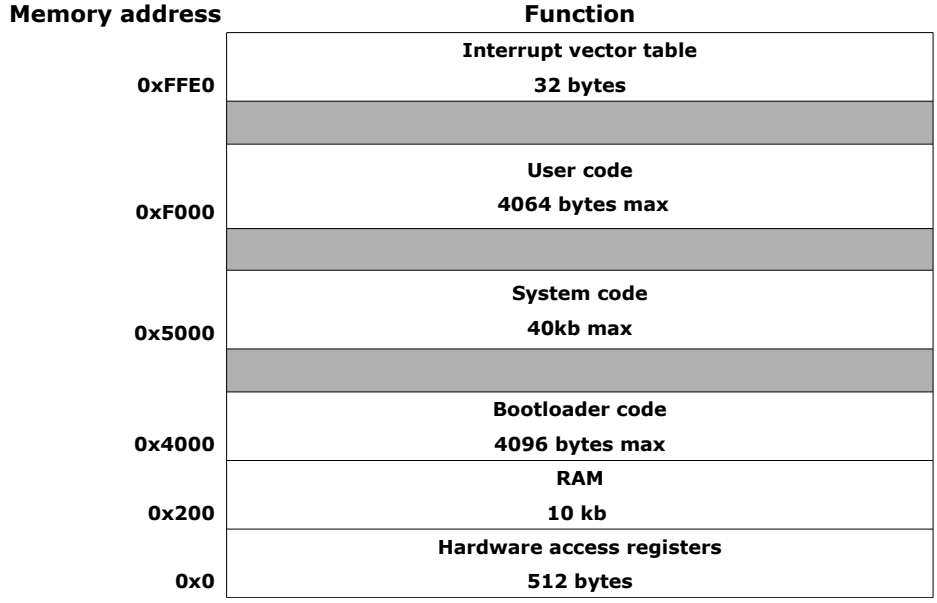


Fig. 5: MansOS memory layout of a *Tmote Sky* application compiled with runtime reprogramming support

Partial reprogramming is supported for energy-efficiency purposes. 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 part without changing the other. In MansOS, partial reprogramming is implemented through address space separation of system and user code (Fig. 5). 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. 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.

4.5 Usability and portability

Although ease-to-use is difficult to quantify, we nevertheless believe that it is an important property of WSN operating systems. Same goes for portability.

MansOS is multi-platform in the sense of supporting multiple hardware platforms (Tmote Sky, Arduino, Zolertia Z1 and more) and multiple architectures (MSP430 and Atmel AVR). MansOS provides platform-independent API for digital communication protocols (UART, SPI, I^2C) and MCU pin configuration. Therefore sensor, external memory and other peripheral drivers can be designed using platform-independent routines, allowing the same driver to be reused among multiple platforms and applications. The communication proto-

cols are provided in both hardware and software versions using unified API. The version to be used is selected at compile time, allowing to reuse peripheral drivers without modification.

MansOS is also multi-platform in the sense of multiple development host OS. While often neglected, quick and easy installation of a WSN OS is an important aspect of its usability. Therefore, MansOS provides self-executable Windows installer and a single Debian package as installation options.* This solution is more lightweight than using number of packages in different formats provided by TinyOS, or the virtual OS image provided by Contiki.

MansOS brings portability further: it is not tied to a particular platform, and also is not tied to a particular development environment. In particular, for MSP430 architecture multiple compilers are supported: *msp430-gcc* versions three and four, as well as IAR MSP430 compiler.

MSP430 GCC is a popular open-source solution for building MSP430 programs. It is powerful, full-featured compiler and can be easily integrated with the GNU debugger (GDB) and `make`. However, in contrast to its x86 version, the MSP430 code generator is not very mature. During development of MansOS we have met all kinds of problems: starting from MSP430 MCU hardware multiplication being used improperly, to generation of incorrect packet field access code, to a skipped instruction in the generated object file causing the program to abort, ending with linker optimizing the `main()` function away.

IAR Embedded Workbench IDE includes a commercial MSP430 C/C++ compiler and debugging options. Compared to open-source alternatives, it supports more MSP430 MCU models. So far we have had no problems with the code generation (possibly because IAR has been used much less than GCC). The main drawbacks of IAR compiler are the severe limitations of all evaluation editions.

In order to provide a visual programming option, we developed MansOS integrated development environment (IDE; Fig. 6). It supports visual and example-based programming, both frequently used by novice and embedded system programmers. The IDE is included in MansOS installation options, thus making the OS easily accessible even to inexperienced users. The IDE uses GCC as the default compiler, and has options for a single-click WSN mote programming, including remotely programming multiple motes at once.

5 Evaluation

This section describes the evaluation of application source code size, binary code and RAM usage, and resource usage by threads. MansOS is compared to three other operating systems: Contiki, TinyOS, and Mantis. Comparison with LiteOS is not performed, since the OS does not run on TelosB platform. We note that a brief evaluation of a few other aspects of MansOS was given in preceding sections.

* Available at `http://mansos.net`.

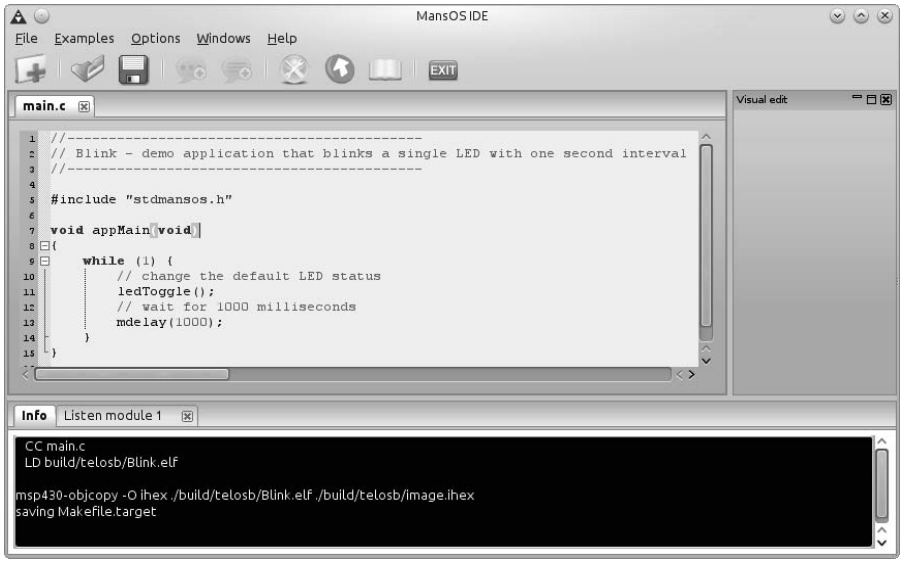


Fig. 6: MansOS IDE

5.1 Source code organization

For evaluation purposes four programs are implemented in MansOS, using both event-based and thread-based approach, as well as in Contiki, TinyOS and Mantis:

- *loop* – the simplest application: execute OS initialization code and then enter an endless loop;
- *radio tx* – transmit 100 byte radio packets periodically;
- *radio rx* – continuously listen for radio packets;
- *combined* – periodically sample sensors, toggle a LED, and transmit the sampled data to radio.

Source code for the *extended combined* application's event-based implementation in MansOS is given in Listing 2. The implementation is extended with external-flash logging.

First we compare source code size for the *combined* application in all five implementations (Fig. 7). The size is evaluated excluding comments and empty lines. Compared to other WSN OS, MansOS allows to write applications with the same functionality using shorter code. This is an important usability benefit of the system, because shorter code is more easy to understand and manage (at least when the complexity is the same). In contrast, large source codes size in TinyOS signal a potential usability problem with this OS. We point out that even though TinyOS applications are written in a different programming language (*nesC*), the abstraction level of the code is roughly the same: they are both high level languages. Further analysis is required to determine whether the complexity per line is small enough in TinyOS to balance out the additional code size.

Listing 2 Example MansOS application

```
#include <stdmansos.h>
#include <hil/extflash.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
    extFlashWrite(extFlashAddress, &packet, sizeof(packet));
    extFlashAddress += sizeof(packet);
    // 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);
}
```

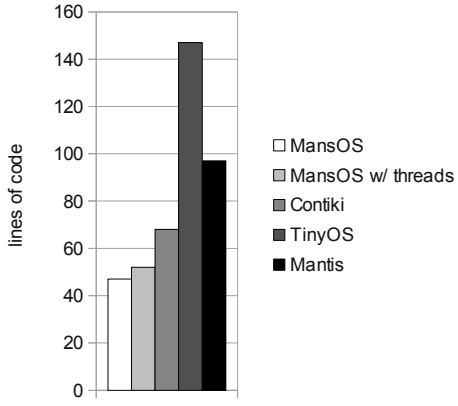


Fig. 7: Source code size comparison for the *combined* application (samples sensors and sends to radio)

5.2 Binary code size

Perhaps more important results are obtained by evaluating binary code sizes (Fig. 8). The source code is compiled for TelosB platform, using MSP430 GCC 4.5.3 compiler. For MansOS, `-O` optimization level is turned on (the default), since higher optimization levels historically have led to broken code. For other OS, their respective default optimization levels are used.

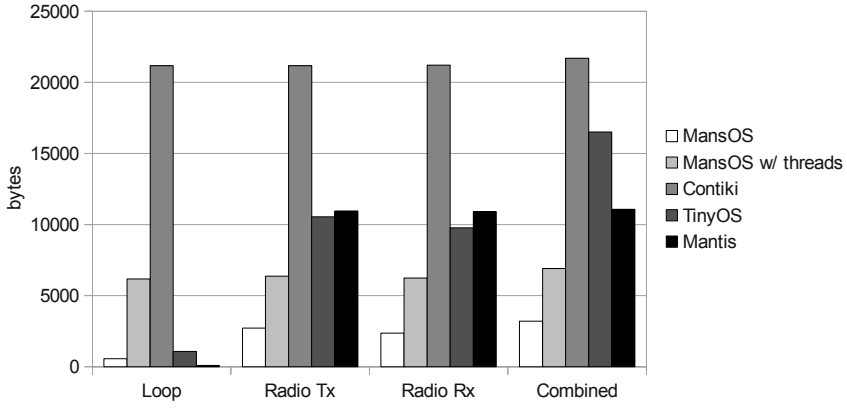


Fig. 8: Application binary size comparison for the *combined* application

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 566 bytes in MansOS (without threads).

Three of four WSN OS analysed try to reduce binary code size in some way. MansOS: by using the configuration mechanism, Mantis: by building separate components as libraries and linking them together, TinyOS: by topologically sorting all functions in source files and pruning unused ones from the final binary

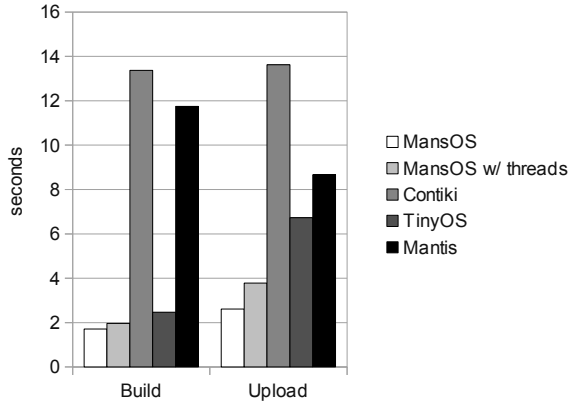


Fig. 9: Application upload time comparison for the *combined* application

code. Only Contiki pays no attention to this problem and demonstrates the worst results of all OS.

Larger binary code size in TinyOS are 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.

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 compiled components is detrimental to the usability of the core system, since the number of inter-component dependencies grows too fast.

Table 2: Flash memory usage in the *extended combined* application, bytes

	No threads	**With threads**
Radio	1394	1644
Kernel	702	720
Flash	454	454
USART & SPI	446	496
Arch & platform	308	370
ADC	182	182
User code	138	188
LEDs	38	38
Library routines	2	132
Threads	0	686
Total	**3270**	**4966**

Shorter binary code size means tangible benefits to the WSN OS user. Firstly, energy requirements in reprogramming are directly proportional to the code size,

Table 3: RAM usage in the *extended combined* application, bytes

	No threads	With threads
User code	14	4
Kernel	10	14
Radio	8	8
USART & SPI	8	8
Flash	2	2
ADC	2	2
Arch & platform	0	0
Threads	0	36
Total	**44**	**74**

if full reprogramming is used. Even though all OS allow some kind of partial reprogramming, full is still required when core parts of the system are changed. Secondly, smaller code leads to shorter development times, as putting the program on the mote becomes faster (Fig. 9). 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). 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.

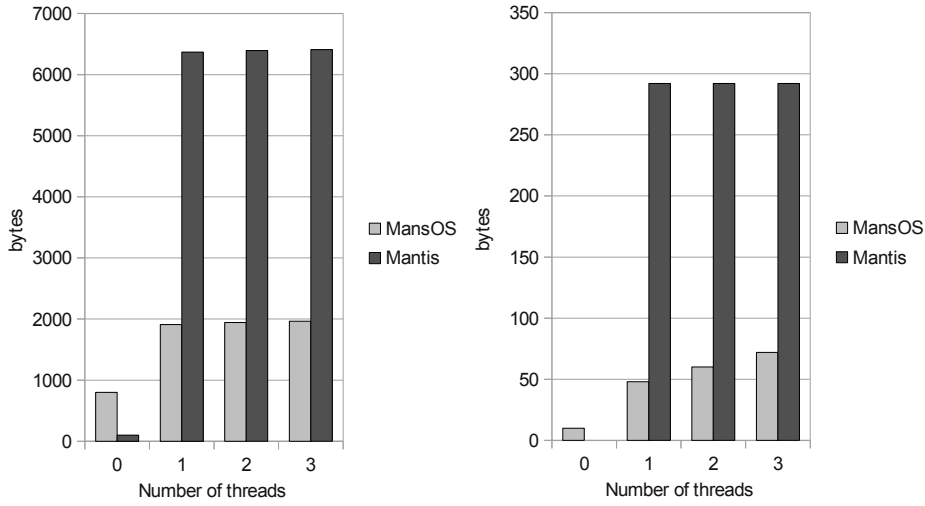
RAM usage is given without including memory allocated for stacks (256 bytes for each thread by default). Even though comparatively large amount of memory is used in this way, it would seldom cause problems for real applications, because code memory, not RAM, is the scarcest resource on Tmote Sky. This is evidenced by the example application (Table 2 and 3), because it uses proportionally more of total code memory (4966 bytes of 48 KB) than of total RAM ($74 + 256$ bytes of 10 KB).

5.3 Thread implementation

Thread implementation is compared with Mantis, because of the four operating systems considered only MansOS and Mantis include preemptive multithreading by default. Both include support for theoretically unlimited number of threads, although in MansOS the upper bound must be specified in 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.

Two distinctive features of MansOS thread implementation become apparent (Fig. 10):

- lower resource requirements. The scheduler used in MansOS is simpler, e.g. it has no separated queues for ready and sleeping threads. This trade-off is justified by the low number of threads typical in a WSN application,



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

Fig. 10: Component usage comparison of a threaded application. *PB*: priority based scheduling, *RR*: round-robin based scheduling.

which allows the scheduler to process all threads in each context switch. Furthermore, in WSN applications the user threads can be expected to be cooperative, so the fairness of the scheduler is not a critical requirement. Finally, thread hierarchy in MansOS (user threads cannot preempt kernel) allows to reduce the number of locks required for thread-safe design.

- better adaptation. Mantis requirements are constant (the flash usage changes are due to longer user code), MansOS requirements are flexible and depend on the number of threads used;

For technical details it should be noted that Mantis example applications uses 128 byte stacks by default. Our selection of 256 byte stacks 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. 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 (Fig. 10b) is used. Finally, we clarify that flash usage differences in Mantis are due to user code changes only.

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. 11). Round-robin based scheduling is more costly, partially because 32-bit *last-time-run* values are used instead of 16-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 ex-

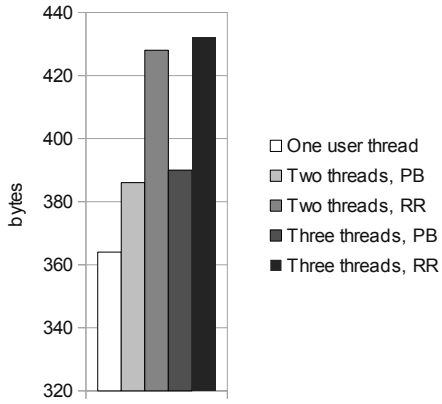
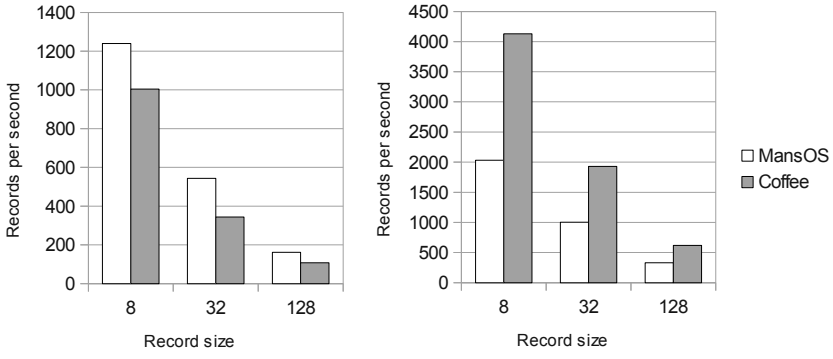


Fig. 11: Thread resource usage comparison: flash memory used by `schedule()` function.

ecution time. However, in all cases the space requirements can be easily satisfied by a typical WSN mote.

5.4 File system

The MansOS file system is evaluated against the Contiki file system, Coffee [15], because both of them have support for the TelosB platform's flash chip. Both are compiled with the GCC 4.6 compiler and the same optimization level.



(a) Write **(b)** Read

Fig. 12: File system write and read throughput

The most important function of a WSN node's storage system is the logging of sensor data. The data typically consist of constant-sized samples. To reflect this, the file systems are tested for how many data records of fixed size they can handle in a unit of time. The results for different record sizes are shown in Fig. 12a (write throughput) and Fig. 12b (read throughput).

The MansOS file system has better performance at writing data, but is slower than Coffee at reading data. Better write performance stems from the fact that

Coffee spends additional time relocating data as the file grows. The read performance of the MansOS file system could be impacted by additional logic layers that increase function call overhead. The read and write speed, at any rate, exceeds typical WSN requirements (from 1 to 100 measurements per second) by a far margin.

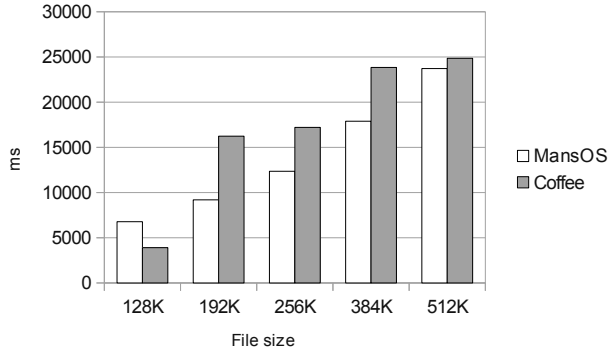


Fig. 13: File system write times

The file systems are also examined for the time it takes to write a file of a certain size: this shows how good the system is at allocating free space and handling files whose size is not known *a priori* (Fig. 13). The time taken by the MansOS file system scales linearly with size of files. Coffee demonstrates a more irregular pattern, which can be explained by that Coffee doubles the size of a file and copies its data once the file capacity is exceeded. Due to the same reason, Coffee cannot handle files that are larger than $\approx 66\%$ of the flash chip capacity (unless the planned file size is told before data are written to the file), while the MansOS file system allows allocating the whole chip to a single file.

6 Conclusions

We have described MansOS, a portable and easy-to-use operating system for wireless sensor networks and resource constrained embedded devices. MansOS is a feature-complete WSN OS with well-structured code. Compact binary code allows MansOS to avoid flash memory overuse problems that are especially prominent in Contiki.

Compared to LiteOS, MansOS is more portable, as it has logical separation between architecture and platform-specific code and the rest of the system.

Compared to Mantis, MansOS has lighter weight threads, as well as separation between the kernel thread and user threads, which in turn facilitates the design of the rest of the system. Locking is often not required, as user threads have no privilegies to preempt the kernel thread.

Compared to Contiki, MansOS is more modular, which in turn leads to lower resource usage overhead, as a MansOS application can use only those components it actually needs. The configuration system reduces both the number of

files that are compiled and the size of binary code, therefore usability is improved, as time taken to build and upload application becomes shorter. Furthermore, using MansOS means less run-time overhead, because module selection and function binding are done at compile time, not at execution time. Finally, MansOS provides a platform-independent (as much as possible) *implementation* of preemptive threads, complete with scheduler and thread-local variables, while Contiki gives only an *interface* of such a model.

Compared to TinyOS, MansOS is more approachable to users without WSN programming knowledge, especially if they are experienced in C programming, because MansOS includes support for multithreaded execution model and is written in plain C. Application source code tends to be significantly shorter as well, with no large obvious increase of complexity per code line, which means that programs written in MansOS are easier to understand and manage because of improved readability.

The OS is friendly to users regardless of their previous WSN programming experience. On one hand, MansOS is targeted towards novice users: it is easy to install, since Windows installer and Debian package are provided, and easy to start using, since MansOS IDE supports visual programming and example-based programming paradigms. On the other hand, MansOS offers a powerful and flexible configuration system for power users and developers of new hardware platforms. The system allows including and excluding specific features, even the MansOS kernel itself, thus providing seamless integration with existing user code and not forcing use of any resource-hungry features.

At the moment MansOS is evaluated in several environmental monitoring projects. In addition, the OS is used to teach WSN programming courses at University of Latvia as an alternative to TinyOS. Our future plans include more field tests of the OS, and support of new hardware platforms, including Intel 8051-compatible architectures.

Acknowledgements

We want to thank European Social Fund for financial support, grant Nr. 2009/-0219/1DP/1.1.1.2.0/APIA/VIAA/020. Work done by Janis Judvaitis on MansOS IDE and other parts of the system is gratefully acknowledged, as are all other MansOS developers.

References

1. Arduino, `http://arduino.cc/`
2. Atmel Corporation: Atmel AVR 8- and 32-bit Microcontrollers. `http://www.atmel.com/products/microcontrollers/avr/default.aspx`
3. Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., Han, R.: MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. Mobile Networks and Applications 10(4), 563–579 (2005)

4. Cao, Q., Abdelzaher, T., Stankovic, J., He, T.: 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. pp. 233–244. IEEE Computer Society, Washington, DC, USA (2008)
5. Dai, H., Neufeld, M., Han, R.: ELF: An Efficient Log-Structured Flash File System. In: Proceedings of the 2nd Conference on Embedded Networked Sensor Systems (SenSys). pp. 176–187. ACM (2004)
6. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. Local Computer Networks, Annual IEEE Conference on 0, 455–462 (2004)
7. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: Proceedings of the 4th international conference on Embedded networked sensor systems. pp. 29–42. SenSys '06, ACM, New York, NY, USA (2006)
8. Durvy, M., Abeillé, J., Wetterwald, P., O'Flynn, C., Leverett, B., Gnoske, E., Vidales, M., Mulligan, G., Tsiftes, N., Finne, N., Dunkels, A.: Making sensor networks IPv6 ready. In: Proceedings of the 6th ACM conference on Embedded network sensor systems. pp. 421–422. SenSys '08, ACM, New York, NY, USA (2008)
9. Handziski, V., Polastre, J., Hauer, J., Sharp, C., Wolisz, A., Culler, D.: Flexible Hardware Abstraction for Wireless Sensor Networks. In: Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005) (2005)
10. Klues, K., Liang, C., Paek, J., Musăloiu-e, R., Levis, P., Terzis, A., Govindan, R.: TOSThreads: Thread-Safe and Non-invasive Preemption in TinyOS. In: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09). pp. 127–140. ACM (2009)
11. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: An operating system for sensor networks. In: Ambient Intelligence. Springer Verlag (2004)
12. Libelium: Waspmote: The Sensor Device for Developers. `http://www.libelium.com/products/waspmote/`
13. Moteiv Corporation: Tmote Sky: Low Power Wireless Sensor Module. `http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf`
14. Texas Instruments: MSP430™ Ultra-Low Power 16-Bit Microcontrollers. `http://www.ti.com/msp430`
15. Tsiftes, N., Dunkels, A., He, Z., Voigt, T.: Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In: Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN). pp. 349–360 (2009)
16. Zolertia: Z1 Platform. `http://www.zolertia.com/ti`