Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 4, April 2014, pg.250 – 262

RESEARCH ARTICLE

OPTIMIZED KERNEL IMPLEMENTATION OF MANSOS

M.Prabhu[1], V.Rajaram[2]

[1]Department of IT, Sri Venkateswara College Of Engineering, India

[2]Department of IT, Sri Venkateswara College Of Engineering, India

[1] m.prabhu16@gmail.com; [2] vrajaram@svce.ac.in

Abstract— Operating system support for Wireless sensor networks (WSN) plays a major role in building scalable distributed applications that are efficient and reliable. Over the years, we have seen a variety of Operating systems emerging in the sensor net community to facilitate developing WSN applications. The design of operating system for WSN is a challenging task. In the proposed model optimized kernel model for portable and easy-to-use WSN operating system has a smooth learning curve for users with C and UNIX programming experience is presented. The OS features a configuration model that allows reducing application binary code size and building time. In contrast to other Wireless sensor networks operating system, MansOS provides both event-based and threaded user application support, including a complete lightweight implementation of preemptive threads

Keywords— WSN; embedded os; os

I. 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. it is 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 SenspireOS 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.

II. **MANSOS ARCHITECTURE**

An easy way to comply with the conference paper formatting requirements is to use this document as a template and simply type your text into it.

A. ABSTRACTION MODEL

As one of our design goals is minimization of e ort 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 (Figure.3.1) 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 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 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.

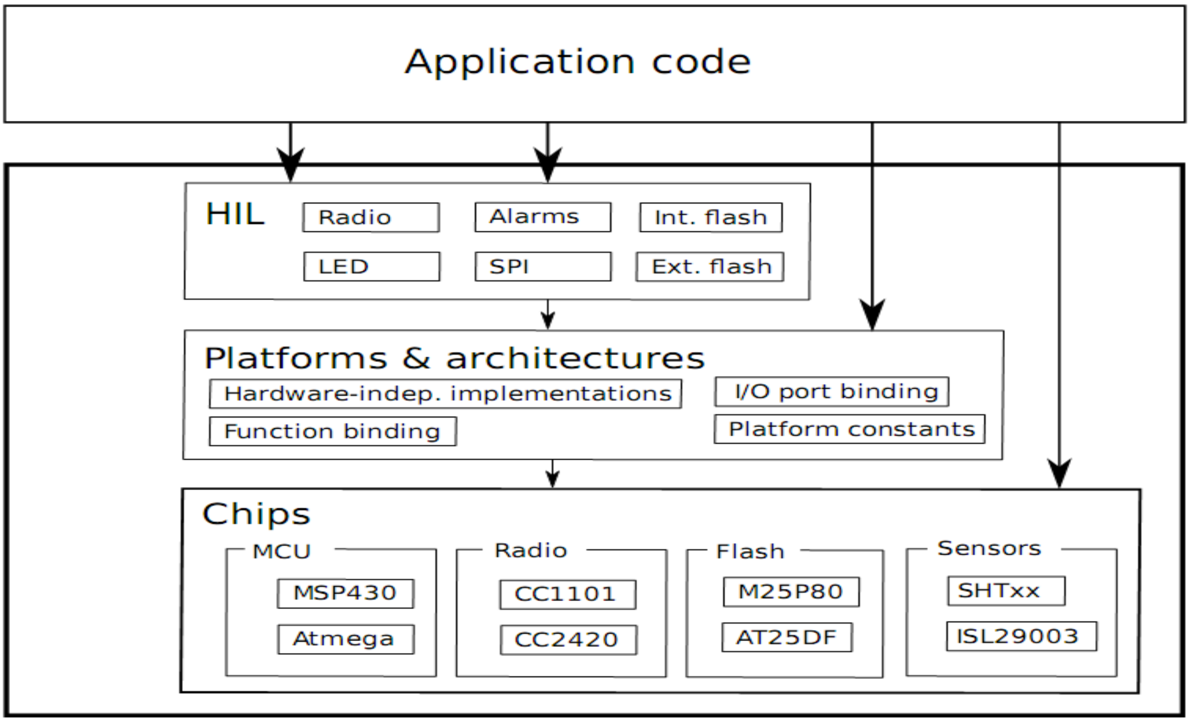

FIGURE 1: MANSOS COMPONENTS AND ABSTRACTION LAYERS

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

	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%**

Table 1: Source Code Size Break down With Regard To Mansos Components, Lines Of Code (Excluding Comments and Empty Lines)

B. 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 IC 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, IC) typically used for communication with the peripherals. Atleast, 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 IC protocol versions are used on different peripheral devices (such as light sensors). The hardware support for IC 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 times tamped. 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.

C. 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].

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 co de. 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 frequently 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 hierarchical: 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 discarding 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.

III. 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.

A. 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.

B. EXECUTION MODELS

Two application execution models are used in WSN OS:

• Event-based (asynchronous);

• Thread-based (synchronous).

Event-based model is simpler and requires fewer 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 lengthy 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 di cult to trace, stack over ow 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.

1) 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 call-back 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 10ms 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 co de 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.

2) THREADED EXECUTION

Thread implementation in a WSN OS can be simplified if two observations are taken into account. First, the number of threads typically required by a WSN application is small. 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 observation 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.

Stack over flow 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 over flow 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.

IV. 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.

A. *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 detecting errors without reading the data twice. The overall division of flash memory into smaller elements is shown in Fig. 3.2.

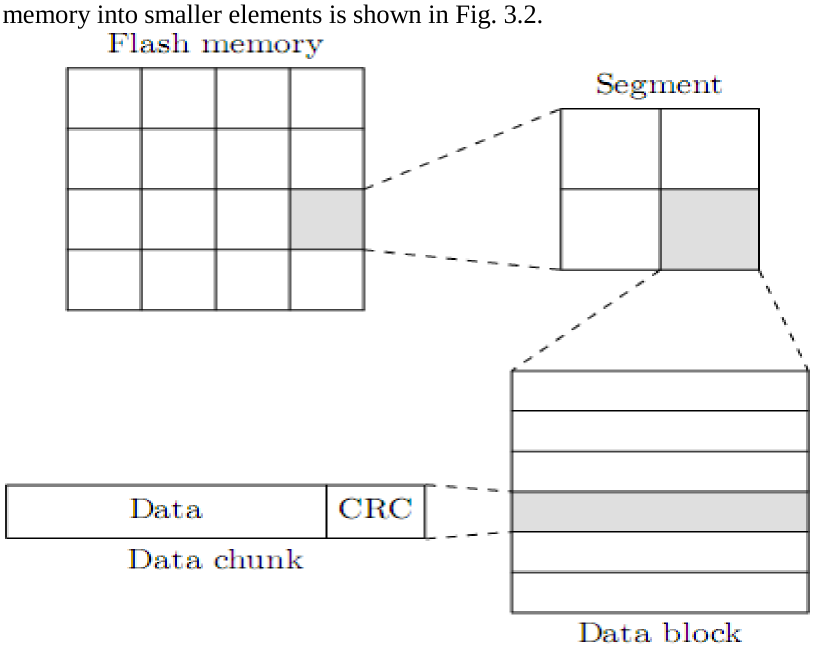

FIG.2: STRUCTURAL ELEMENTS IN THE FLASH MEMORY

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.

B. DATA BLOCK MANAGEMENT

Information about data blocks is held in the *block table*, a bitmap containing the current state of each data block (Fig.3.3). The block table is small enough to be stored in the WSN node's EEPROM memory, where it can also be easily updated.

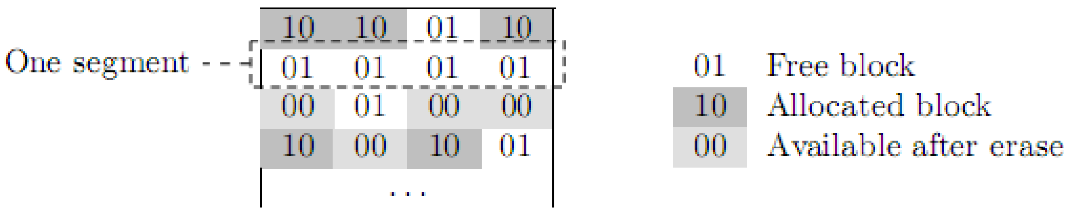

FIG.3: BLOCK TABLE FORMAT

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 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.

C. Control structures

File entries are kept in the root directory, which is also stored in the Electrically erasable programmable read only 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.

V. **RESULTS AND DISCUSSION**

A. *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 . 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 co de. 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 co de size in TinyOS signals a potential usability problem with this OS. We point out that even though TinyOS applications are written in a di erent 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

1) *SAMPLE CODE*

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);

}
```

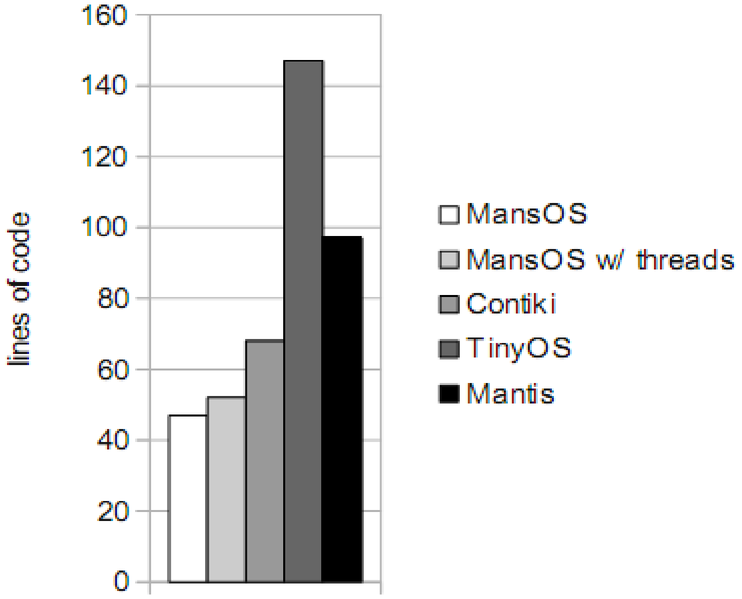

Figure.4: Source Code Size Comparison For The *Combined* Application

complexity per line is small enough in TinyOS to balance out the additional code size.

B. *BINARY CODE SIZE*

Perhaps more important results are obtained by evaluating binary code sizes (Fig. 4.2). The source co de 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.

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 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 co de. 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.

	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**

Table 2: Flash memory 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**

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

Shorter binary code size means tangible benefits to the WSN OS user. Firstly, energy requirements in reprogramming are directly proportional to the code size, if full reprogramming is used. Even though all OS allow some kind of partial re-programming, full is still required when core parts of the system are changed. Secondly, smaller co de leads to shorter development times, as putting the program on the mote becomes faster (Fig. 4.3). 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 4.1). While using more resources in general, the threaded version leads to shorter user code and smaller RAMS 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 4.1 and 4.2), because it uses proportionally more

of total code memory (4966 bytes of 48 KB) than of total RAM (74 + 256 bytes of 10 KB).

VI. CONCLUSION

I 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 privileges to preempt the kernel thread.

Compared to SenspireOS, 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 co de, 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 SenspireOS 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 co de 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

REFERENCES

[1] 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)

[2] 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)

[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 E cient Log-Structured Flash File System. In: Proceedings of the 2nd Conference on Embedded Networked Sensor Systems (SenSys). pp. 176–187. ACM (2004)

[6] Atmel Corporation: Atmel AVR 8- and 32-bit Microcontrollers. http://www.atmel.com/products/microcontrollers/avr/default.aspx

[7] 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)

[8] 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)

[9] Klues, K., Liang, C., Paek, J., Musaloiu-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)

[10] 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)

[11] Libelium: Waspmote: The Sensor Device for Developers. http://www.libelium.com/products/waspmote/

[12] Moteiv Corporation: Tmote Sky: Low Power Wireless Sensor Module. http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[13] Texas Instruments: MSP430™ Ultra-Low Power 16-Bit Microcontrollers. http: //www.ti.com/msp430

[14] Tsiftes, N., Dunkels, A., He, Z., Voigt, T.: Enabling Large-Scale Storage in Sensor Networks with the Coee File System. In: Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN). pp. 349–360 (2009)

[15] Zolertia: Z1 Platform. http://www.zolertia.com/ti