An Operating System Abstraction Layer for Portable Applications in Wireless Sensor Networks

Ramon Serna Oliver
Chair of Real-Time Systems
TU Kaiserslautern
Germany
serna_oliver@eit.uni-kl.de

Ivan Shcherbakov
Chair of Real-Time Systems
TU Kaiserslautern
Germany
shcherbakov@eit.uni-kl.de

Gerhard Fohler
Chair of Real-Time Systems
TU Kaiserslautern
Germany
fohler@eit.uni-kl.de

ABSTRACT

Portability of software modules is a major concern in application development for Wireless Sensor Networks (WSN), stressed by the typical lack of resources in embedded systems. Abstractions of the hardware platform which are introduced by the operating system (OS) allow the development of modules which can be reused in new applications. However, the lack of standards in this domain, restricts the chances to achieve efficient portability to those systems running on very similar platforms (e.g. same OS).

In this paper, we present an Operating System Abstraction Layer (OSAL), which unifies the OS architecture and establishes a common API across multiple OS. Portability of applications is effectively granted thanks to a common set of primitives, which are independent of the underlaying OS and its particular architecture.

We highlight the efficiency of the OSAL as well as detailed description of its main features and design considerations. We have implemented the OSAL on top of two well known OS and performed extensive evaluations, which show that it effectively reduces portability efforts at the expenses of minimal run-time overhead as well as negligible increase of memory footprint.

Keywords

Abstraction layer, API, Operating Systems, OS, Portability, Wireless Sensor Networks, WSN, OSAL.

1. INTRODUCTION

The increasing popularity of Wireless Sensor Networks (WSN) [1] and their expansion to new application domains demand fast development and quick integration of software and hardware components. Portability of software components is a key issue, which becomes essential to achieve fast deployments of large and complex systems [2].

Hardware platforms have evolved from simple boards with minimal sensing and communication capacities to complex systems equipped with customized hardware modules and advanced architecture. Hence, operating systems (OS) play an important role in abstracting the underlaying hardware architecture, which is exposed to the software applications through the OS API. New hardware components can be added to the platform (e.g. new sensor attached to an SPI bus), existing hardware can be upgraded (e.g. exchange of radio transceiver) or even the complete hardware platform may be exchanged without requiring major modifications at the application level.

Unfortunately, there is a broad number of OS targeting the domain of embedded sensor networks (e.g. [3], [4], [5], [6]), providing a common set of core functionalities and services. The choice for a particular deployment may depend on external factors, support of the chosen platform architecture (i.e. CPU family), availability of drivers for sensors and/or communication buses, existence of a network stack or particular protocols (e.g. 802.15.4), or special characteristics of the OS (e.g. real-time scheduler).

In this paper, we present an Operating System Abstraction Layer (OSAL), which embraces the management of hardware configurations as well as access to specific set-points that represent a major hook to performance trade-offs. OSAL defines a subset of OS primitives which satisfies the basic application builder's requirements but at the same time, remains simple to cover the API of most targeted OS. It is designed to resemble a subset of a POSIX [7] OS API, with the motivation of reducing the learning-curve as well as the preference for a neutral reference without specific features from any particular platform.

We implemented OSAL on top of two well-known OS for sensor networks developed in C: MANTIS OS [5] and FreeRTOS [4]. At the moment, OSAL is designed to cover the majority of thread-based OS , which offer better portability support with respect to other event-based systems. Further work may explore the extension to event-driven OS (e.g. TinyOS).

Our analysis shows that OSAL introduces minimal runtime overhead and negligible increase of memory footprint while achieving efficient portability of applications. Typical OS primitives to create tasks, mutexes and timers experience no additional code size overhead (0 bytes) in MANTIS OS and very low in FreeRTOS (i.e. less than or equal 12 bytes). With respect to the footprint overhead of initialized and non-initialized memory, OSAL has no impact at all in these primitives.

The rest of this paper is organized as follows: Section 2 briefly explores related work in this area. Section 3 describes the abstractions and principles of OSAL, while Section 4

SAC'10 March 22-26, 2010, Sierre, Switzerland.

presents an extensive evaluation based on the current implementations. Finally, Section 5 concludes the paper.

2. RELATED WORK

Little work has been done towards the unification of the existing variety of OS targeting sensor networks. Well established standards like POSIX [7], including specific profiles for embedded systems, are too complex for the limited availability of resources in typical sensor platforms.

Among the existing OS, we highlight the following for their relevance to the current state-of-the-art:

Mantis OS

MANTIS OS [8] aims at a low memory footprint, easy to program architecture, and support for preemptive multiple threads. It follows a prioritized threaded programming model similar to classic POSIX with a scheduler based on priorities and a round-robin policy for those threads with same priority.

The OS consists of a kernel with integrated scheduler, a command server and device driver system. It supports mutual-exclusion and semaphores. It also integrates a low-level communications stack for serial and radio communication interfaces, including a MAC layer and a device abstraction layer that provides uniform access to devices. MANTIS OS is mostly implemented in C and several ports exist for different platforms.

FreeRTOS

FreeRTOS [4] falls in the category of a kernel rather than a full operating system as generally understood. It supports, among other features, prioritized and preemptive threads, interrupt service routines (ISR), mutexes and queues and dynamic memory allocation.

FreeRTOS favors simplicity and portability over optimization. Nearly all the code is written in C, with only a few assembler functions. The scheduler follows either a prioritized preemptive or cooperative scheduler policy, depending on its configuration. In the former, tasks with the same priority share CPU time in a round robin fashion, while in the latter context switches only occur if a task blocks or yields.

Contiki

Contiki [9] is an operating system designed for memory-constrained environments, such as sensor networks. It is built around an event-driven kernel [10], and features dynamic loading and unloading of individual programs and services. It supports a full TCP/IP stack via the μIP library, as well as programming abstractions via protothreads. Contiki is implemented in C and has been designed to be easily portable to new platforms.

TinyOS

TinyOS [6] is a popular run-time system specifically designed for networked sensors. Its wide adoption is one of its main strengths together with the availability of a rich library of networking and application components.

TinyOS implements an event-driven execution paradigm where every execution is triggered by some external event interrupt. It provides physical device abstractions as a conventional OS does. The programming model exposed by NesC incorporates the event-driven execution, the flexible concurrency model, and the component-oriented application design of TinyOS. However, it requires the programmer to adopt a very careful programming policy which may jeopardize the ability of reusing code in various deployments.

Virtual Machines

An alternative approach to achieve abstraction of the OS and platform architecture is to develop on top of virtual machines (VM). Squawk, Maté [11], Sentilla [12] and other VM are build to target sensor networks and small embedded systems. Despite the benefits of VM in terms of quick deployment and maintainability, we argue that the overhead in terms of execution and memory utilization represent a serious inconvenience for complex deployments.

3. OSAL OVERVIEW

The establishment of proper control mechanisms for the hardware platform into software frameworks brings up a number of portability issues. OSAL [2] has been designed to address these issues and diminish the conflicts between different software and hardware platforms. In particular, it addresses the discrepancies among different OS with respect to their functional API, hardware configuration mechanisms, resource management and handling of peripherals.

OSAL is an abstraction layer designed to be placed on top of an OS, which translates system primitives from the original operating system into an unified API. Thus, application builders are able to use a common API which has the advantage of dramatically reducing portability efforts in later deployments. Our experience shows that after the first implementation, porting OSAL to new platforms requires a relatively small effort compared to a large scale deployment.

OSAL achieves minimal footprint and execution overhead by using advanced compilation tools and techniques. Among others, function level linking, in-line functions and extensive use of preprocessor macros to provide light function wrappers for the OS native primitives.

Main achievements of the OSAL include the unification of basic data types and structures, a common API to for system primitives and return codes, extension or implementation of non-compliant or missing primitives, and abstraction of specific OS and hardware initialization procedures.

3.1 OSAL API

OSAL API covers the basic needs to develop applications in the domain of sensor networks. We highlight the most representative parts of its design with appropriate examples for each case below.

OS initialization

Start-up initialization is one the most system-dependent features in any OS. For example, FreeRTOS leaves this responsibility to the entry function, which normally creates some tasks before calling `vTaskStartScheduler()` to trigger the OS scheduler. This primitive does not return, and hence the initial task itself is permanently suspended.

In contrast, MANTIS OS initializes the scheduler before calling the application-defined entry function (`void start()`), where any additional required task can be created.

OSAL defines a initialization function (`wos_main()`) which always runs after the scheduler has been initialized. Any task created in this function will be scheduled immediately.

Mantis OS

```
void start() {
   // Application initialization code
   // Application main thread
}
```

FreeRTOS

```
void MainThread() {
   // Application main thread
   vTaskDelete(xTaskGetCurrentTaskHandle());
}
int main() {
   vPortInitialize();
   // Application initialization code
   xTaskCreate(MainThread, NULL, main_stack_size,
               NULL, main_priority, NULL );
   vTaskStartScheduler();
   return 0;
 }
```

OSAL

```
wos_status wos_main(void) {
   // Application initialization code
   // Application main thread
   return MAKE_STATUS(WOS_SUCCESS);
}
```

Table 1: System initialization of an empty application in MANTIS OS, FreeRTOS and OSAL

Table 1 summarizes the initialization procedure for the three systems.

Tasks

The OSAL API unifies the way in which parameters are given to the OS and the particularities of each individual system.

Creating a task using OSAL is done with the primitive `wos_task_create(thread_func, NULL)`, where the second parameter can be either `NULL` or a pointer to an initialized `task_attr_t` structure that allows specifying stack size and priority explicitly.

Synchronization

Primitives to ensure mutual exclusion between concurrent tasks are frequently used in embedded OS. OSAL provides support for mutex, semaphores and message queues, which cover most application requirements with respect to synchronization and inter-task communication.

Table 11 shows the primitives for mutex handling of the of the two analyzed OS for comparison with the OSAL API. Note that the FreeRTOS synchronization API has several critical drawbacks (e.g. impossibility to immediately wake up a thread of the same priority which was waiting for a semaphore). For this reason, we developed an additional *Advanced Synchronization Framework* with similar syntax which solved these issues.

Mantis OS

```
mos_mutex_t mtx;
mos_mutex_init(&mtx);
mos_mutex_lock(&mtx);
mos_mutex_unlock(&mtx);
```

FreeRTOS

```
xQueueHandle mtx = xQueueCreateMutex()
xSemaphoreTake(mtx);
xSemaphoreGive(mtx);
```

OSAL

```
wos_mutex_t mtx;
wos_mutex_init(&mtx);
wos_mutex_lock(&mtx);
wos_mutex_unlock(&mtx);
```

Table 2: Synchronization primitives in MANTIS OS, FreeRTOS and OSAL

Software Timers

Software timers are of special importance in sensor network systems as they allow implementation of time-outs as well as periodic execution of functions, which are essential to any networking protocol. Efficient implementations use hardware timer interrupts to callback a given timer function. This, however, implies a number of limitations due to the execution within an interrupt context (ISR). Namely, that interrupts are disabled while a software timer handler is running and the "current task" context is undefined, which causes that blocking functions cannot be used.

The following example shows how software timers are handled in OSAL:

```
timer_t timer;
wos_timer_create(&timer, period, hdl_funct, context);
wos_timer_destroy(&timer);
```

Note, that FreeRTOS does not originally support software timers. We implemented this functionality before porting the OSAL API on top of FreeRTOS.

3.2 Message handling

Efficient handling of messages is vital to sensor applications. OSAL unifies the most relevant aspects regarding radio transmissions of messages. Namely, message structures, buffers and radio API.

Packet buffers

As different OS use different structures for packet buffers (e.g. some of them are fixed-size, some support variable length), OSAL provides a unified preprocessor macro allowing to declare such a structure, which specifies its name and the desired size.

For example, OSAL provides procedures to declare packet buffers and structures both for sending and receiving packets (i.e. `DECLARE_PACKET(name, max_size)`). Particular constrains may appear due to native restrictions on the underlaying OS. For instance, the parameter `max_size` must

be checked and a compilation error raised if it exceeds the maximum supported size by the OS.

Radio API

The radio API is a subset of the OSAL which abstract internal structures and management of data packets. Packet buffers can be both fixed or variable size, and can contain (or not) several internal fields. The goal of the radio API is to abstract platform-specific details from the user and to provide a transparent interface for dealing with packets.

Particular attention has been given to the buffer management for radio transmission and reception. For example, MANTIS OS copies incoming packets into a fixed system-maintained buffer (i.e. a packet is always received in the same place and then can be copied by the application). OSAL provides a preprocessor macro, which enables direct access to the system buffers, avoiding costly and unnecessary memory operations.

MANTIS OS provides relatively inconvenient API for sending and receiving radio packets. A packet should be contained in a fixed-size buffer structure and the `size` field should be set directly by the primitive caller. A typical scenario involving sending some fixed-structure packets using MANTIS API is the following:

```
struct VerySimpleMessage {
   unsigned long OrderNumber;
};
comBuf buf;
buf.size = sizeof(VerySimpleMessage);
((VerySimpleMessage *)buf.data)->OrderNumber = n;
com_send_IFACE_RADIO(&buf);
```

OSAL provides a more convenient object-oriented API for sending radio packets, reducing the previous code to the following:

```
EnclosingPacket<VerySimpleMessage> buf;
buf->OrderNumber = n;
Radio::SendPacket(buf);
```

Moreover, OSAL hides all OS-specific low-level details, such as fixed-size buffers, which enables straight forward portability among different OS.

3.3 Build system

Developing almost every embedded application starts with setting up the build environment. One of the goals of the OSAL is to simplify build system-related tasks as much as possible.

The steps required to build an application differ on the two analyzed OS. Under MANTIS OS, the user needs to create an `automake` file to generate a `Makefile`, which must be re-generated when new sources are added. No built-in debug/release configuration support is provided, despite this can be done by editing the generated `Makefile`. Generally, a typical `automake` file consists of $3 - 4$ lines.
FreeRTOS does not provide a build system itself. Hence, users should manually create `Makefile` and specify all involved sources and build flags. A typical FreeRTOS-based application `Makefile` consists of around 50 lines.

We explored the possibilities of `GNU make` to provide a simple building system for OSAL. As a result, we created a number of scripts which dramatically reduce the build system-related complexity, as shown in the following `makefile` example to build a simple application:

```
WOSMAKE_ROOT = ../../../../makesystem
include $(WOSMAKE_ROOT)/Makeapp.lazy
```

In this case, the binary file name and the source list is automatically generated based on current directory name and contents. Alternatively, it is possible to specify this information explicitly:

```
WOSMAKE_ROOT = ../../../../makesystem
wosapp_objects = my_obj1.o my_obj2
wosapp_image = my_binary
```

Note that the OSAL build system automatically supports switching between MANTIS and FreeRTOS, as well as between `DEBUG` and `RELEASE` configurations, requiring no makefile or source file modifications.

4. EVALUATION

The OSAL has been successfully implemented on top of MANTIS OS and FreeRTOS as part of the EU funded project WASP [13] with the internal name *WASP OS API* (WOS). Complete documentation of the design and implementation process is publicly available in [14] under the *Work Package 3* public deliverables. Documentation regarding the WOS API is periodically updated in [14].

4.1 Code and memory footprint overhead

We evaluated the overhead on the executable code size and the memory footprint of initialized and non-initialized memory. We identify the overhead of each basic primitive and extrapolate the run-time overhead based on the composition of these measurements. Our implementation ensures that no additional run-time overhead exists due to the abstraction layer.

We created a number of sample applications to easily identify when and why overhead was generated comparing four OS configurations: MANTIS OS, MANTIS OS + OSAL, FreeRTOS, and FreeRTOS + OSAL.

The following tables show the evaluation of of executable size, initialized (Mem-I) and non-initialized (Mem-NI) memory in bytes for each individual functionality.

OS initialization overhead

Table 4 shows the evaluation of the start-up initialization functions in Table 1. In this example we compare the byte sizes for the compiled code in `RELEASE` and `TRACE` modes. In the latter, OSAL initializes the *tracing framework* (mutex and some buffers), responsible of the additional overhead compared to the former.

The difference between the two modes is due to the availability of debug tracing as, e.g. `wos_dbg_print("Bug!");`. This allows inserting debugging lines that only produce binary code in `TRACE` mode, resulting in no overhead when compiling in `RELEASE` mode.

Our evaluation shows a minimal overhead of 22 bytes in Mantis OS and 40 bytes in FreeRTOS due to the initialization process with much lower overhead on the memory footprint in `RELEASE` mode. Note that the impact of the `TRACE` configuration is relatively small, enabling the use of debugging primitives at a reasonable cost.

Starting from the synchronization example, we will use the OSAL initialization function for both raw RTOS and

Mantis OS

```
#include <mos.h>
#include <msched.h>
#include <led.h>
#define DEFAULT_STACK_SIZE 128

void thread1(void) {}
extern "C"
void start(void) {
   mos_thread_new( thread1, DEFAULT_STACK_SIZE,
                   PRIORITY_NORMAL);
}
```

FreeRTOS

```
#include "FreeRTOS_all.h"

void thread2(void *) {
   vTaskDelete(xTaskGetCurrentTaskHandle());
}
void thread1(void *) {
   FreeInitialThreadStack();
   xTaskCreate(thread2, NULL, 64, NULL, 1, NULL);
   vTaskDelete(xTaskGetCurrentTaskHandle());
}
int main() {
   FreeRTOS_InitializeForWASP();
   xTaskCreate(thread1, NULL, 64, NULL, 1, NULL);
   vTaskStartScheduler();
}
```

OSAL

```
#include <wos/task.h>
#include <wos/led.h>

void thread1(void) {}
wos_status wos_main(void) {
   wos_task_create(thread1, NULL);
   return MAKE_STATUS(WOS_SUCCESS);
}
```

Table 3: Simple application example creating one empty task

	Size	Mem-I	Mem-NI
MANTIS	9048	20	534
MANTIS+OSAL(R)	9056 (+8)	20	534
MANTIS+OSAL(T)	9070 (+22)	20	544 (+10)
FreeRTOS	4448	46	114
FreeRTOS+OSAL(R)	4474 (+24)	46	114
FreeRTOS+OSAL(T)	4488 (+40)	46	116 (+2)

Table 4: Overhead introduced by OSAL in system initialization functions (see Table 1) compiled in different modes: `TRACE` (T) and `RELEASE` (R).

OSAL-based builds to filter out the constant initialization overhead and to track only the changes implied by using OSAL API instead of raw RTOS API.

	Size	Mem-I	Mem-NI
MANTIS	9066	20	534
MANTIS+OSAL	9074 (+8)	20	534
FreeRTOS	4480	46	114
FreeRTOS+OSAL	4510 (+30)	46	114

Table 5: Overhead introduced by OSAL in task creation functions (see Table 3).

	Size	Mem-I	Mem-NI
MANTIS	9088	20	544
MANTIS+OSAL	9088	20	544
FreeRTOS	4518	46	116
FreeRTOS+OSAL	4526 (+8)	46	116

Table 6: Overhead introduced by OSAL in task creation functions with original OS primitives (see Table 3).

Task handling overhead

To evaluate the overhead added to the task handling primitives we consider the code shown in Table 3. Table 5 shows that the overhead is mostly the same as for previous sample which is caused by the initialization functions.

To distinguish between startup-related overhead and other types of overhead, we change the FreeRTOS and MANTIS samples to use the OSAL initialization while directly calling the original OS primitives. The modified code is shown in Table 10.

Note that when FreeRTOS is used, OSAL wraps all thread functions in an own function that calls `vTaskDelete()` after a thread function has returned. This produces a constant 6 byte overhead plus 2 bytes for each task being created.

Further examples in the rest of this paper, do not include the initialization overhead in order to provide a better overview of the the isolated overhead of each evaluated functionality.

Synchronization overhead

We evaluate the overhead of synchronization primitives by comparing the footprint sizes of a relatively small program that performs iterative mutex testing, as shown in Table 11.

Table 8 shows that OSAL does not introduce any overhead for MANTIS OS, and the only overhead for FreeRTOS are the already mentioned 6 bytes plus 2 additional bytes for each task created, which are due to the task function wrapping. Therefore, the synchronization API itself does not produce any additional overhead, neither in code size nor memory footprint.

Timers overhead

To evaluate the overhead introduced in software timers, we compare the footprint sizes for a sample application registering three timer handlers which increase one of three static variables. Again, as OSAL functions are simple inline wrappers around the native OS calls, no overhead is produced as shown in Table 7.

	Size	Mem-I	Mem-NI
MANTIS	10842	20	679
MANTIS+OSAL	10842	20	679
FreeRTOS	6974	46	210
FreeRTOS+OSAL	6974	46	210

Table 7: Overhead introduced by OSAL in a sample timer application.

	Size	Mem-I	Mem-NI
MANTIS	11324	32	639
MANTIS+OSAL	11324	32	639
FreeRTOS	6974	58	164
FreeRTOS+OSAL	6986 (+12)	58	164

Table 8: Overhead introduced by OSAL in synchronization functions (see Table 11).

	Size	Mem-I	Mem-NI
MANTIS	11498	22	633
MANTIS+OSAL	11498	22	633

Table 9: Overhead introduced by OSAL in a sample message sending application (see Table 12).

Mantis OS

```
#include <wos/task.h>
#define DEFAULT_STACK_SIZE 128

void thread1(void) {}
wos_status wos_main(void) {
   mos_thread_new(thread1, DEFAULT_STACK_SIZE,
                  PRIORITY_NORMAL);
   return  MAKE_STATUS(WOS_SUCCESS);
}
```

FreeRTOS

```
#include <wos/task.h>

void thread2(void *) {
   vTaskDelete( xTaskGetCurrentTaskHandle());
}
wos_status wos_main(void) {
   xTaskCreate(thread2, NULL, 64, NULL, 1, NULL);
}
```

Table 10: Modified simple application example creating one empty task with original OS primitives

Radio API overhead

With respect to the radio API, Table 12 shows a simple application sending a sequence of short messages. Since FreeRTOS does not directly support radio interfacing, at this time, we only evaluate this feature with MANTIS OS. Table 9 shows that the usage of OSAL primitives do not increment the footprint, hence incurring in no additional overhead due to OSAL.

```
#include <wos/task.h>
// ...
static wos_mutex_t s_TestMutex;
static int s_Busy = 0;
static int s_Error = 0;
static int s_Iter = 0;
static int s_Threads = 0;
#define MAX_SLEEP_T  100

void thread_body(void) {
   int i;
   wos_mutex_lock(&s_TestMutex);
   s_Threads++;
   wos_mutex_unlock(&s_TestMutex);
   for (i = 0;;i++) {
      wos_mutex_lock(&s_TestMutex);
      if (s_Busy) {
         WOS_LED_ON(RED);
         wos_dbg_print("Bug!");
         s_Error = 1;
      }
      s_Busy = 1;
      wos_sleep(((unsigned)rand()) % MAX_SLEEP_T);
      s_Iter++;
      s_Busy = 0;
      wos_mutex_unlock(&s_TestMutex);
   }
}
wos_status wos_main(void) {
   wos_mutex_init(&s_TestMutex);
   wos_task_create(thread_body, NULL);
   wos_task_create(thread_body, NULL);
   wos_task_create(thread_body, NULL);
   for (;;) {
      wos_dbg_printf("Iteration: %5d;
                     threads: %d; error: %d\n",
                     s_Iter, s_Threads, s_Error);
   }
   return MAKE_STATUS(WOS_SUCCESS);
}
```

Table 11: Sample OSAL application using synchronization primitives.

4.2 Evaluation overview

Table 13 summarizes the additional overhead introduced by OSAL for each of the analyzed functionalities. Note that the overhead, measured in bytes, is accounted for each of the referred functions in isolation (e.g. ignoring the overhead due to initialization). The total additional overhead of a complex application depends on the number of tasks and other resources created, although our analysis shows that it is negligible with respect to the overall footprint of a real application.

```
struct VerySimpleMessage {
   unsigned long OrderNumber;
};
using namespace WOS::Radio;
wos_status wos_main(void) {
   wos_radio_init();
   EnclosingPacket<VerySimpleMessage> buf;
   for (unsigned long n = 1;;n++) {
      buf->OrderNumber = n;
      wos_dbg_printf("Sending... (%ld)", n);

      WOS_LED_ON(GREEN);
      Radio::SendPacket(buf);
      wos_dbg_printf("done\n");
      WOS_LED_OFF(GREEN);
      wos_sleep(100);
   }
   return MAKE_STATUS(WOS_SUCCESS);
}
```

Table 12: Sample OSAL application sending a sequence of short messages containing increasing numbers.

	MANTIS OS			FreeRTOS		
Function	(a)	(b)	(c)	(a)	(b)	(c)
Initialization	+8	0	0	+24	0	0
Tasks	0	0	0	+8	0	0
Mutex	0	0	0	+12	0	0
Timers	0	0	0	0	0	0
Messages	0	0	0	-	-	-

Table 13: Overview of overhead introduced by OSAL (in bytes) in (a) code size, (b) initialized memory, and (c) non initialized memory.

5. CONCLUSIONS

Portability of application-level code is a major concern in fast and efficient developments for sensor networks. We presented an Operating System Abstraction Layer (OSAL) to reduce the portability efforts of software applications between platforms, independent of the running OS. We claim that such design reduces dramatically the required efforts related to portability of application code, and effectively enables the re-utilization of software components in later deployments.

We performed an extensive evaluation on two implementations on top of MANTIS OS and FreeRTOS, which are currently being exploited in the development of a full network stack for wireless sensor networks. Our tests show that the introduced overhead is negligible while the common API guarantees that binaries can be compiled on both platforms without introducing any change in the source code. Our experience show that similar approaches can be followed with other thread-based operating systems, incurring in little relative effort compared to a large deployment.

Future and on-going work includes the extension of the FreeRTOS port to support the radio API as well as further evaluations on top of additional platforms. Additional work to extend OSAL to event-driven OS is under consideration.

A complete description of the OSAL API is periodically updated in [14].

6. ACKNOWLEDGMENT

This work is partially financed by the European Commission under the Framework 6 IST Project "Wirelessly Accessible Sensor Populations (WASP)".

7. REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[2] A. Schoofs, M. Aoun, P. van der Stok, J. Catalano, R. Serna Oliver, and G. Fohler, "On enabling portable and time-controlled wireless sensor network applications," in *Proceedings of the 1st International Conference on Sensor Networks Applications, Experimentation and Logistics (SENSAPEAL09)*, Athens, Greece, September 2009.

[3] "Contiki OS," 2009. [Online]. Available: http://www.sics.se/contiki/

[4] "FreeRTOS," 2009. [Online]. Available: http://www.freertos.org/

[5] "MantisOS," 2009. [Online]. Available: http://mantis.cs.colorado.edu/

[6] "TinyOS," 2009. [Online]. Available: http://www.tinyos.net/

[7] M. A. Rivas and M. G. Harbour, "Evaluation of new posix real-time operating systems services for small embedded platforms," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[8] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications (MONET) Journal*, vol. 10, no. 4, pp. 563–579, 2005.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[10] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "An experimental comparison of event driven and multi-threaded sensor node operating systems," in *Third IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PERSENS2007)*, I. C. S. Press, Ed., White Plains, USA, March 2007.

[11] P.Levis and D.Culler, "Maté: A tiny virtual machine for sensor systems," *SIGOPS Oper.Syst.Rev.*, vol. 36, 2002.

[12] "Sentilla." [Online]. Available: http://www.sentilla.com

[13] "EU Framework 6 IST Project "Wirelessly Accessible Sensor Populations" (WASP)," 2006-2010. [Online]. Available: http://www.wasp-project.org/

[14] "The WASP OS API." [Online]. Available: http://rts-wiki.eit.uni-kl.de/WASP/index.html