A Framework for Time-Controlled and Portable WSN Applications

Anthony Schoofs[1], Marc Aoun[2], Peter van der Stok[2], Julien Catalano[3], Ramon Serna Oliver[4], and Gerhard Fohler[4]

[1] CLARITY: Centre for Sensor Web Technologies, Dublin, Ireland
anthony.schoofs@ucdconnect.ie
[2] Philips Research, Eindhoven, The Netherlands
{marc.aoun,peter.van.der.stok}@philips.com
[3] Enensys Technologies, Rennes, France
julien.catalano@enensys.com
[4] University of Kaiserslautern, Kaiserslautern, Germany
{serna_oliver,fohler}@eit.uni-kl.de

Abstract. Body sensor network applications require a large amount of data to be communicated over radio frequency. The radio transceiver is typically the largest source of power dissipation; improvements on energy consumption can thus be achieved by enabling on-node processing to reduce the number of packets to be transmitted. On-node processing is facilitated by a timely control over process execution to sequence operations on data; yet, the latter must be enabled while keeping high-level software abstracted from both underlying software and hardware intricacies to accommodate portability to the wide range of hardware and software platforms. We investigated the challenges of implementing software services for on-node processing and devised constructs and system abstractions that integrate applications, drivers, time synchronization and MAC functionality into a system software which presents limited dependency between components and enables timely control of processes. We support our claims with a performance evaluation of the software tools implemented within the FreeRTOS micro-kernel.

Keywords: Wireless sensor networks, portability, temporal control.

1 Introduction

The use of low-cost and intelligent physiological sensor nodes, capable of sensing, processing, and communicating one or more vital signs to a monitoring station is the next step for health monitoring. Wearable health monitoring systems can closely monitor changes and provide feedback to either the remote clinician or the patient himself about life-threatening alerts, as well as to maintain an optimal health status. Because of the scarce resources available on the nodes, achieving both accurate health monitoring and long-life node activity is rendered difficult. Indeed, accurate health monitoring requires continuous acquisition and reasoning on sensor data. A stroke rehabilitation system is an example of sensor-based

N. Komninos (Ed.): SENSAPPEAL 2009, LNICST 29, pp. 126–144, 2010.

system where the amount of data and frequency at which they are transmitted is high [5]. The direct consequence is the intensive use of the Radio Frequency (RF) transceiver, which dominates the power consumption of a node. Power consumption can be reduced by enabling reasoning and processing of data on the node; this offers the opportunity to avoid the transmission of raw data and to exchange only valuable information processed from sensor data. For this type of applications, *enabling software services for on-node processing* is required. Also, in this application where multiple nodes work together for monitoring health, *a common notion of time shared by the nodes* is required to treat, correlate and combine the different types of data. It allows collective signal processing, accurate duty-cycling, data aggregation, and distributed sampling.

SAND nodes [1] are an example of flexible body sensor nodes with a fixed core around which application specific sensors and actuators can be placed. Another motivation of this work is to support this view of flexible nodes by providing a *software system with a fix core around which software components can be seamlessly integrated, and ported to new hardware configurations*. Portability facilitates reuse of existing applications and software for on-node processing in new execution environments. With these goals, we investigated the necessary software services and system abstractions, and the challenges to integrate them without depreciating the system's performance.

Section 2 presents information essential to understand the motivation of this work and related work. In Section 3 we present the software constructs that we devised and the problems solved. We validate our work with a performance evaluation of the software tools implemented within the FreeRTOS micro-kernel in Section 4. Finally Section 5 concludes.

2 Background

2.1 Constraints with Development in Lightweight Software Environments

The following summarizes the usual constraints associated with lightweight or monolithic software environments.

Hardware dependent application code. This depicts a situation where direct accesses to hardware low-level details are included into application software (e.g. writing hardware registers for peripheral control). When hardware interfaces are changed or not fully compatible, additional effort to adapt code deviates application developers from their main application development task, forcing them to dive deep into the hardware details.

No separation of independent processes. One single thread for the program execution forces the application developper to combine independent application processes. For the envisioned application scenario, data acquisition, data processing, and time synchronization would end up correlated, making the code unclear and difficult to debug.

No control on timing. By combining different application processes in one single thread, it becomes difficult to keep accurate control on timing. Changes in one part of the code involve new time delays for other processes that require new timing calculations at each occurrence.

No prioritization of processes. One single thread executes sequentially the different application processes and no priority can be given to a process.

Limited portability. With application specific code, written for a specific hardware configuration and intricated within different processes, very little can be reused directly.

2.2 Challenges to Enabling Temporal Control over Processes and System Abstraction

Timely control over processes on the node requires a clear separation of independent software processes and means of scheduling these processes according to their priority and timing deadlines. Global or distributed timely control over processes additionally requires that a common notion of time is shared by the nodes. That way, they are able to locally schedule and execute processes in global synchrony. A first challenge is then to enable accurate execution of processes on the node, contingently based on a global time knowledge.

Sequencing operations on data requires a close interaction with the underlying system services. This aspect contrasts with the portability needs of algorithms and applications, that should be favored to accommodate the wide range of hardware and software platforms. A second challenge is then to ensure that providing timely control over processes to high-level software does not prevent portability.

Drivers and applications are often provided by different developers, and either need to be developed or ported to the hardware platform. As the software development grows, it is important to bring all the software parts together. Merging implementations exposes the software system to code redundancy and bugs. Therefore, a consistent and reliable system abstraction on which to write high-level software is advocated. A third challenge is to devise an abstraction that enables an optimal usage of the platform (possibly specific) hardware and software mechanisms, while presenting a uniform top application programmer interface (API) hiding those various intricacies.

2.3 Related Work

On the software side little consensus exists about the basic functionalities that should be at the disposal of the application developer. In a sense the situation for software is worse than the situation for the hardware. Where the hardware goals are more or less clear: smaller and lighter with less energy consumption, the software goals are undefined. Many people experiment in isolation with their favoured software paradigm to create answers to isolated problems. Identified problems centre around software constructs, which give access to the hardware

with a low consumption in memory. We aim at providing a software platform on which a given class of applications can be developed, removing the problems of synchronization and many hardware intricacies from the application, as well as enabling on-node processing to lower the overall energy consumption.

Event-based and preemptive multithreading are two approaches to programming resource constrained systems. Event-based operating systems are efficient as they require little memory and processing resources. TinyOS is a very famous open source component-based operating system and platform targeted to wireless sensor networks [10]. It is designed to incorporate rapid innovation as well as operate within the severe memory constraints inherent in sensor networks. It is largely written in C and NesC, a component-based programming language and an extension to the C programming language. Multithreading is not provided in the original version to prevent excessive stack usage and the system bases its operation on run-to-completion semantics. The efficiency comes at the cost of additional programming overhead; connecting results of software modules, keeping track of execution flow, and determining when a specific function ends is the programmer's responsibility. Later work introduced TinyThread, a library for TinyOS that enables true multi-threading on a mote [11]. Task preemption required to meet temporal requirements is not provided, but has been enabled with low memory overhead by Duffy et al. [12]. A second event-driven embedded operating system is Contiki [13]. Contiki uses protothreads, a lightweight form of cooperative threading, implemented as pre-processor macros. Protothreads provide the programmers with a more natural way of formulating their code. Contiki implements implicit network time synchronization. No explicit time synchronization messages are sent: the module relies on the underlying network device driver to timestamp all radio messages, both outgoing and incoming. An average synchronization error of 1.4 ticks was achieved in single-hop network, with one tick equal to 2ms [14]. A high time synchronization accuracy is needed to be able to synchronize the execution of events on separate nodes, and to be able to time-stamp precisely events and sampled data. While TinyOS and Contiki lack native real-time support, FreeRTOS provides pre-emptive scheduling based on task priorities. FreeRTOS is a portable, open source, micro Real Time Kernel [2]. It is responsible for managing system resources, processor, memory and I/O peripherals. FreeRTOS code base is small, and is mostly written in standard C. Each task is assigned a priority and tasks with the same priority share the CPU time in a round-robin fashion. FreeRTOS scheduler is configured as preemptive to answer the real-time behaviour required by the system. FreeRTOS queue mechanism is priority aware for real-time treatment. Tasks and Interrupt Service Routines (ISR) can also communicate using queues, and binary semaphores are implemented as a special case of queues. Similar to Contiki, FreeRTOS does not provide a native 802.15.4 MAC implementation.

In this work we have strong requirements on temporal execution of processes for on-node and distributed processing. We also favor the integration of a generic and standard compliant MAC protocol, as the presented framework should fit the needs of multiple applications scenarios and different hardware configurations.

The aforementioned OS and micro-kernels do not natively provide the software for supporting those requirements. A choice of micro-kernel as the core of our software environment was made. Our choice of adopting FreeRTOS, instead of an event-based OS such as Contiki, has a number of reasons. Application programmers are usually more familiar with traditional multi-threaded programming, in comparison with an event-based environment where an explicit differentiation might exist between requesting operations in one phase, and processing notifications in a second phase. This is in contrast with traditional multi-threaded environments, where the execution flow of individual tasks is easier to follow. Additionally, in an event-based programming model, when combined with run-to-completion semantics, it becomes the responsibility of the application programmer to carefully divide long-running tasks into multiple smaller entities to ensure a timely flow of execution of the system as a whole.

Software portability is achieved with a consistent and extensive abstraction of the underlying system: abstraction of the microcontroller hardware details and system resources. Much work has been done in researching portability for application development in wireless sensor networks. The authors of [15] have successfully achieved with the MAC Layer Architecture (MLA) a component-based approach for developing MAC protocols. Code is reused and intricacies of hardware platforms are hidden to the developer. Our objective is different; we aim at facilitating the development of applications, device drivers and middleware. MAC protocols are generally modeled as a communicating state machine, where each state represents a different phase in the communication protocol. The execution of the state machine is supported by the underlying software system. Our work on the MAC does not aim at producing hardware dependent code reused by different MAC protocols; we aim at breaking the influence of software constructs, supporting the MAC state machine execution, on higher layer processes' timing control and portability. In [16], a three-layer hardware abstraction architecture has been introduced with the objective of increasing portability and simplifying application development. Although such design facilitates the building of platform independent applications, abstraction of hardware intricacies needs to be conjugated with an abstraction of the underlying software system.

2.4 Software Architecture Overview

In the presented framework, timely control over processes is enabled via the use of micro-kernel services, time synchronization, and a task-based IEEE-802.15.4 MAC implementation. Software portability is achieved with a hardware peripheral abstraction and an operating system (OS) API, built around the micro-kernel. Figure 1 depicts the devised software framework architecture. The 6LoWPAN adaptation layer and UDP/IP protocol stack are described in [8]. The task synchronization component enables synchronized execution of processes on multiple nodes. We show in Section 4 how this service for distributed node processing has been integrated within the framework.

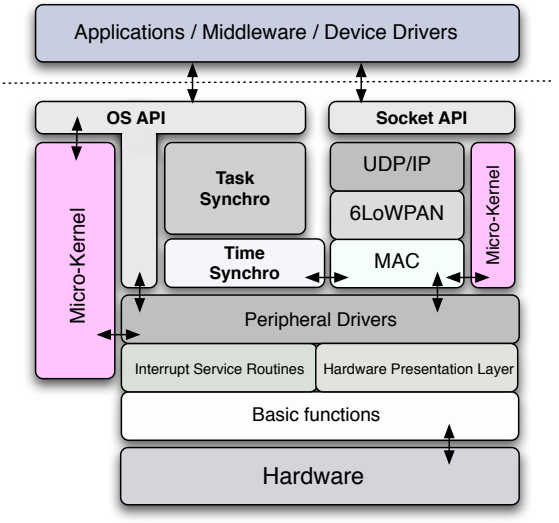

Fig. 1. Software framework architecture

3 Architecture

3.1 A Micro-kernel as the Core of the Software System

The choice of a micro-kernel is motivated by the memory and resource constraints of low-power microprocessors and by the flexibility in term of functionalities that are supported. Every system designer can use the same base, the micro-kernel, and add or remove components as required by the application. For wireless sensor networks applications, multi-threaded kernels offer the possibility to run fixed-priority tasks which guarantee the execution of processes that need real-time behaviour. The type of applications devised for wireless sensor networks requires a small number of tasks, which limits the memory and latency overhead. A multi-thread environment helps the programmer separate unrelated code, and breaks the timing and event dependency. By having independent applications in different tasks, it becomes also easier to read the code and debug it. We used FreeRTOS micro-kernel to implement and validate our software constructs. It is responsible for managing system resources, processor, memory and I/O peripherals.

3.2 Achieving Timely Control over Processes

Reducing the MAC Influence on Application Process Execution

Motivation. The Medium Access Control (MAC) is an important piece of software which impacts on the architecture of the total node software. The IEEE 802.15.4 standard [18] has defined a physical and MAC layer for low bit-rate and low-power consumption, to enable communication within nodes. Transmission of

data or command frames relies on a Carrier Sense Multiple Access with Collision Avoidance (CSMA-CA) protocol to schedule the communication. CSMA-CA is implemented using units of time called backoff periods, which are generated via a periodic event (e.g. timer periodic tick) on the microcontroller. Most 802.15.4-compliant transceivers have only a small subset of 802.15.4 features built in hardware and MAC protocol mechanisms including CSMA-CA are implemented in software on the node's microcontroller. Therefore, both MAC and applications share the same processing power. Generally, 802.15.4 MAC software is implemented as interrupt-driven, where the MAC state machine is executed within interrupt service routines, executed every 320μs on MAC tick interrupts (e.g. [19]). The regular backoff-slot tick and the lengthy software execution within the ISR impose unnecessarily constraints to the whole software system. Ticks are happening even when RF communication is not required, and thus increases overhead. Besides, once an interrupt-based MAC implementation is adopted, the application is often developed as one background task removing all possibilities of independently executing tasks. A later switch to a task oriented micro-kernel becomes very difficult. Also, when interrupt nesting is not supported by the microcontroller of a new hardware platform, an interrupt-driven implementation becomes unusable. We propose a new architecture based on a preemptive task execution environment. With the proposed architecture, MAC software is activated by the micro-kernel scheduler, providing a solution where MAC software is not executed by an ISR, but by a micro-kernel task.

A task-based implementation. Micro-kernel tasks are scheduled on the basis of the periodic tick interrupts of the micro-kernel timer. Our design takes a dual scheduling approach, where a second timer dedicated to the MAC is started for handling RF communication: one timer generates OS ticks for the micro-kernel scheduling whereas a second one generates ticks for the MAC scheduling, only when needed. With more powerful timer engines, one single timer can generate the two ticks. The interest is to be able to adapt dynamically the tick frequency to the application, to avoid useless overhead and reduce power consumption, and to clear the influence of the MAC on the software system by removing the execution of MAC software from ISRs. With this approach two timer bases are maintained and MAC software is executed within a micro-kernel task, removing the architecture constraints of interrupt-based implementations.

General aspects on implementation. We have implemented an 802.15.4 MAC onto the FreeRTOS micro-kernel following the task-based architecture. In the micro-kernel -based implementation of the MAC, the MAC is considered as an application whose duty is to drive the transceiver chip to be compliant with the 802.15.4 standard. Accordingly, the MAC will be run by one FreeRTOS task, which we call the FreeRTOS MAC task. Currently, the FreeRTOS MAC task has the highest priority of all FreeRTOS tasks, in order to respect timing constraints related to network wireless communication. Two FreeRTOS binary semaphores have been created to control the operations for packet transmission (TX) and packet reception (RX). Initially and repeatedly, the FreeRTOS MAC task checks the `FRTOS_MAC_semaphore` used for controlling TX operations, as shown below:

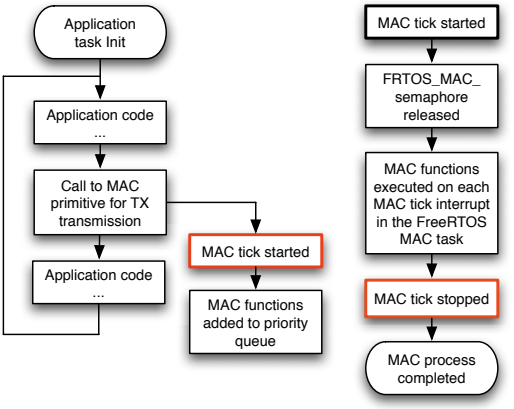

Fig. 2. Task-based MAC process execution for a packet transmission

```
portTASK_FUNCTION(FreeRTOS_MAC_Task,pvParameters){
for(;;) {
//Wait for the semaphore to become available
if(xSemaphoreTake(FRTOS_MAC_semaphore,portNO_BLOCK)){
```

When packet transmission is not required, the FreeRTOS MAC task cannot take the `FRTOS_MAC_semaphore` semaphore. Next, the FreeRTOS MAC task checks the `FRTOS_MAC_FIFOP_handle_semaphore` in case a RX packet is to be retrieved, as shown below:

```
if (xSemaphoreTake(FRTOS_MAC_FIFOP_handle_semaphore, portBLOCK)){
```

In case no packets are received, and after a time defined by the `portBLOCK` parameter, control returns and the FreeRTOS MAC task loops back to the first `if` condition.

Higher-level protocols that need to access the MAC use the 802.15.4 primitives. Transmission of data happens during units of time called backoff slots. For that purpose MAC primitives called by higher layers will add MAC requests to one of four priority queues. The FreeRTOS MAC task executes the corresponding functions by taking the oldest request from the highest priority non-empty queue, and executes the corresponding function, starting on a backoff slot boundary. The execution of MAC functions is state machine based on a task information structure. This structure contains information on the currently active functions and the associated packets. As long as the priority queues are filled, the MAC timer is active and generates interrupts every 320μs. On interrupts from the IEEE 802.15.4 chip, the MAC timer ISR releases the `FRTOS_MAC_semaphore` thus liberating the blocked FreeRTOS MAC task which executes the MAC functions according to the contents of the task information structure, as shown in Figure 2.

On packet reception, the transceiver generates interrupts at the microprocessor to indicate the reception of an 802.15.4 packet. The FIFOP interrupt signals that a complete frame has been received. When the FIFOP interrupt occurs, the corresponding ISR releases the `FRTOS_MAC_FIFOP_handle_semaphore` that blocks the FreeRTOS MAC task. The ISR terminates and the micro-kernel scheduler checks whether new micro-kernel tasks were enabled by the interrupt. The semaphore released by the ISR unblocks the FreeRTOS MAC task. The latter retrieves the payload from the transceiver.

Providing a Common Notion of Time throughout the Network

Motivation. Peer sensor nodes lack a common notion of time. Since timing is of importance for coherently time-stamping packets in the network, performing data fusion and correlating sensory data, a time synchronization procedure is needed to provide distributed sensor nodes with the same notion of time. In order to synchronize the clocks in our testbeds, we integrated a time synchronization component based on the FTSP protocol [4]. In FTSP, nodes in a network synchronize their clocks to that of a single time master. Multihop time synchronization is supported by requiring any node synchronized to the master to assume a time master role for its unsynchronized neighboring nodes. FTSP combines accurate MAC-layer time-stamping of time synchronization packets [21] and clock drift rate estimation using linear regression [22]. With the motivation to provide the high timing accuracy required for on-node data processing and distributed sampling, while reducing the associated overhead, we introduced the use of Kalman filtering for estimating the clock drift, which gave interesting insight in the reduction of clock synchronization messages exchange [23].

General aspects on implementation. To achieve time-stamping of time synchronization messages at the MAC layer, we used the Start of Frame Delimiter (SFD) interrupt, specified in the IEEE 802.15.4 standard, as a triggering event for time-stamping. The SFD interrupt occurs at the sender of a time synchronization message and at the receiver of that same message. Neglecting propagation time, these "twin" interrupts can be considered to be simultaneous. A timestamp made at the sender side is included in its corresponding time synchronization message. This is achieved by writing the first part of the message's payload to the CC2420 transmission buffer (TxFIFO), initiating transmission, waiting for the SFD interrupt to occur and the time-stamp to be made, and then writing the time-stamp to the TxFIFO in time before the buffer runs out of bytes to send and the CC2420 signals a buffer underflow. At the receiver side, a second time-stamp, made upon the occurrence of the "twin" SFD interrupt, is associated with the received message. As such, a pair of time-stamps related to the same time synchronization message becomes available at the receiver side. The latter applies linear regression on a set of n consecutive pairs of time-stamps, or alternatively Kalman filtering, to determine the clock drift rate between sender and receiver. Time synchronization is implemented in a FreeRTOS task, and is an optional service available to the application developer. The interrupt subscription mechanism depicted in Section 3.3 keeps time synchronization related

code out of other components e.g. the MAC. The time synchronization task subscribes to the SFD interrupt and defines code to execute whenever the interrupt fires (at the sender and at the receiver). The code related to time-stamping and addition of the time-stamp to the packet is then confined to the time synchronization task, and makes the module non-intruding.

3.3 Achieving Portable Software

We designed a set of abstraction layers that break the application dependency on hardware intricacies and software system.

Abstracting the Hardware Intricacies

Motivation. The motivation behind abstracting the hardware intricacies of a microcontroller is to ease application development, enable code re-use between different applications, optimize hardware interaction, facilitate the portability of device drivers and applications, protect access to hardware peripherals and ease debugging.

Challenges. Abstracting hardware should not compromise efficiency by hiding platform features that may optimize a set of processes. For instance, the 24-bit architecture of the NXP CoolFlux DSP equips it with a SPI interface that allows transfers of up to three bytes per transaction, as opposed to traditional 1-byte transactions. Making use of such feature improves process latency and accompanying energy by reducing the occurrence of micro-kernel context switches on each peripheral interrupt and lowering the overhead of fetching bytes in the memory. Another challenge is the handling of peripheral transactions in hardware without introducing high latency overhead. Hardware handling of peripheral transactions in parallel to software execution prevents the use of blocking calls.

Description. We developed an hardware abstraction with a layered architecture to present a common interface to applications for controlling the underlying hardware. This architecture provides all the necessary functions to communicate with the different hardware blocks. The architecture is given in Figure 3.

The *Basic functions* layer is a very thin layer enabling a way to read and write to the input and output registers of the micro-controller and to redirect hardware interrupts to the *Interrupt Service Routines (ISR)* layer. The *ISR* layer handles the interrupts, and can be used to free a semaphore or post data to a queue using the micro-kernel system interface, operate on the hardware such as acknowledging the interrupt or sending an extra byte, or simply run application code. In order to keep the application code separate from the hardware specific sections of the code, the *ISR* layer provides a subscription mechanism which allows for applications or device drivers to perform custom actions within the interrupt. Applications tasks, device drivers and peripheral drivers can subscribe to a specific ISR and write some code, placed outside the hardware specific code, to be executed when the interrupt fires. The *Hardware Presentation Layer (HPL)* provides a complete and dense interface of all the hardware peripherals to the

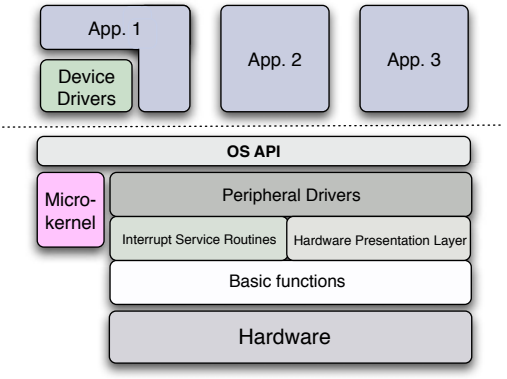

Fig. 3. Hardware Abstraction Architecture

upper layer. This layer is stateless, meaning that it will not keep information about the status of a peripheral. It executes the order from upper layers. This layer, despite the fact that it provides standard functions, is not generic to all the platforms as it contains platform specific options. It is meant to expose all the functions available on the underlying hardware. Finally, the *Peripheral Drivers* layer, using the *HPL*, provides the application or the upper device drivers with a set of functions to use the hardware safely and in interaction with the micro-kernel. In this layer, the state of the interface is kept in a dedicated structure and data and/or events related to a bus or a timer are also recorded in micro-kernel queues or semaphores. The common top interface is the peripheral driver layer interface. The layers underneath are platform specific and should not be exposed to the application layer for portability reasons as well as for the risk of misuse. In order to make use of any platform specific features, the peripheral layer is implemented such that it optimally makes use of the underlying hardware via the *HPL*.

Figure 4 depicts an example where the MAC initiates a packet transmission. In the process, the packet is transmitted to the radio transceiver via the SPI interface before the actual physical transmission. For that purpose, the CC2420 radio driver uses the API of the SPI peripheral driver. The latter passes the first byte of data to the HPL layer, and the remaining bytes are stored in the SPI TX FreeRTOS queue, for later transmission. The byte is eventually transmitted via SPI once written in the SPI data register. On SPI transfer completion, the SPI ISR checks in the TX queue the whether other bytes need to be transmitted. If yes, the next byte is passed to the HPL layer for immediate transmission and so on until the last byte is transmitted.

Abstracting the Software Environment

Motivation. A subset of existing OS are widely used by the WSN community. Porting applications among different operating systems is a time consuming exercise which requires a deep knowledge of both application domain and OS

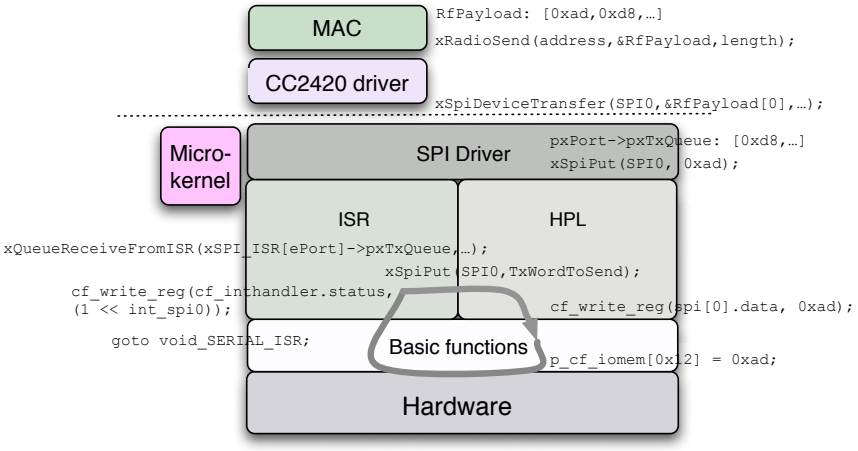

Fig. 4. Example describing the Hardware Abstraction Architecture for a packet transmission

architecture. In the case of WSN, even though most operating systems count with a subset of similar services from an application point of view, they are presented with a different API, thereby making portability harder.

OS Abstraction layer. The definition of proper control mechanisms for the hardware platform into the software framework arise a number of portability issues. The OS abstraction layer [25] (OSAL) exposing an abstraction of the OS API is designed to address these issues and diminish the conflicts between different software and hardware platforms. In particular, it addresses the discrepancies among different OSs with respect to their functional API, hardware configuration mechanisms, resource management and handling of peripherals.

Application builders are offered a common API which abstracts the underlying OS and hardware platform. Hence, portability among different platforms is mostly reduced to the re-implementation of the OSAL without introducing changes to the interface between software and hardware platforms (i.e. OSAL API).

The OSAL API defines a subset of OS primitives which satisfies the basic application builder's requirements but at the same time, remain simple to match most of the target OSs. Use of other native functionalities not covered by the OSAL is discouraged as it violates the principles of portability.

The OSAL embraces the management of hardware configurations and access to specific set-points which represent a major hook to performance trade-offs. The procedure to design the OSAL covers the following steps:

– Selection of primitives done with a concrete analysis of the typical application requirements on the WSN domain as well as further portability considerations.

Table 1. OSAL Profiles

Profile	Description
OSAL.0: Core	Basic system handling and initialization
OSAL.0.1: System	System primitives
OSAL.0.2: Task and scheduler	Task initialization and scheduling primitives
OSAL.0.3: Basic I/O	I/O primitives
OSAL.1: Memory	Dynamic Memory management
OSAL.2: Synchronization	Task synchronization
OSAL.2.1: Mutexes	Mutexes
OSAL.2.2: Message Queues	Message Queues
OSAL.2.3: Signals	Inter-task signaling
OSAL.3: Time	Time handling primitives primitives
OSAL.3.1: Clock	Internal clock handling
OSAL.3.2: Self-suspension	Sleep and delay primitives
OSAL.3.3: Timers	Handling of timers
OSAL.4: Extended I/O	Non-basic I/O primitives
OSAL.5: ELF-loader	Dynamic code loader

- Definition of an API to expose the set of primitives. Our design is based on a POSIX-like[1] API [26] [27] motivated by the aim of reducing the learning-curve as well as the preference of a neutral reference which does not reflex specific features from any platform. Hence, generality prevails over particular features, which is a reasonable price to pay for portability.
- Implementation of the OSAL with a minimal footprint and execution overhead, which is achieved by means of advance compilation tools and techniques.

General aspects on implementation. This approach focuses on WSN multi-task operating systems implemented in C/C++, which covers the majority of available OS in the field. The implementation of low-overhead wrappers for the native OS' primitives is achieved with the support of different programming utilities:

- Macros are very useful for renaming of functions, data types and parameter re-ordering.
- Inline functions are similar to macros in what they can be used for, but require support from the compiler. They introduce less risk of inconsistencies but not always reduce the run-time overhead.
- Function level linking, when supported by the compiler, reduces the size of executables by preventing the linkage of unused functions.

Table 1 shows a list of profiles which subdivides the set of primitives present in the OSAL API. The categorization is done following a functional behavior

[1] POSIX is a well structured standard which covers most of the features supported by any multi-task oriented OS. It is extensively documented and used for many different purposes. Note that one of the several POSIX profiles is defined to target embedded systems. However, it is too complex for the average operating system running on WSN and therefore its adoption to cover the OSAL API is not pursued.

and based on a non-arbitrary analysis of applications in the WSN domain. This design allows the implementation of the OSAL in a progressive manner without loosing the integrity. Profiles are simple and contain dependent functionalities which need to be implemented together but can be discarded as a whole if such functionalities are not required.

4 Experimental Validation

Our software framework has been tested in multiple application scenarios with different requirements in parallel to its development e.g. to develop an home stroke rehabilitation application [6], an emotion sensing application [7], and a herd monitoring application [8]. This has allowed for a validation of the different modules and an extensive testing under various contexts, which generated some refinements of both the architecture and the implementation, leading to a stable and well-tested services. The following gives some details on the services operation and performance.

4.1 Temporal Control

Process execution example. An example of an application task calling the *mcpsDataRequest* MAC primitive to transmit a data packet is illustrated in the following to depict the execution of processes within the micro-kernel environment. Figure 5 shows the challenges and benefits of multi-tasking application tasks with the kernel task implementation of the 802.15.4 MAC. The *Idle* task is scheduled whenever no other tasks are ready to run. The respective mechanisms indicated by numbers are explained for the indicated three moments.

1. The FreeRTOS application task runs from the OS tick interrupt and initiates a packet transmission by calling the *mcpsDataRequest* MAC primitive. A packet transmission requires MAC functions (e.g. CSMA/CA) to be executed on 802.15.4 backoff slots. For that purpose, *mcpsDataRequest* initializes the MAC timer to generate ticks every 320μs and sets the FreeRTOS MAC task as ready-to-run at the highest priority to execute MAC functions on MAC tick interrupts. The MAC primitive, executed by the application task, also adds the *mtxScheduleTransmission* function into the low priority MAC function queue to be run by the FreeRTOS MAC task to initiate the transmission, and blocks until the end of the transmission.
2. When the MAC tick interrupt happens, the `FRTOS_MAC_semaphore` is released. On return from the interrupt, the FreeRTOS scheduler checks whether new tasks have been enabled. Because the `FRTOS_MAC_semaphore` has been released, the FreeRTOS MAC task is enabled and has the highest priority. It executes the *mtxScheduleTransmission* function, as shown on Figure 5. The *mtxScheduleTransmission* function is a state machine function, where each successive state is executed on MAC tick interrupts. The first state of *mtxScheduleTransmission* is executed, and the FreeRTOS MAC task is suspended until the next MAC tick. Because the application task is blocked, the

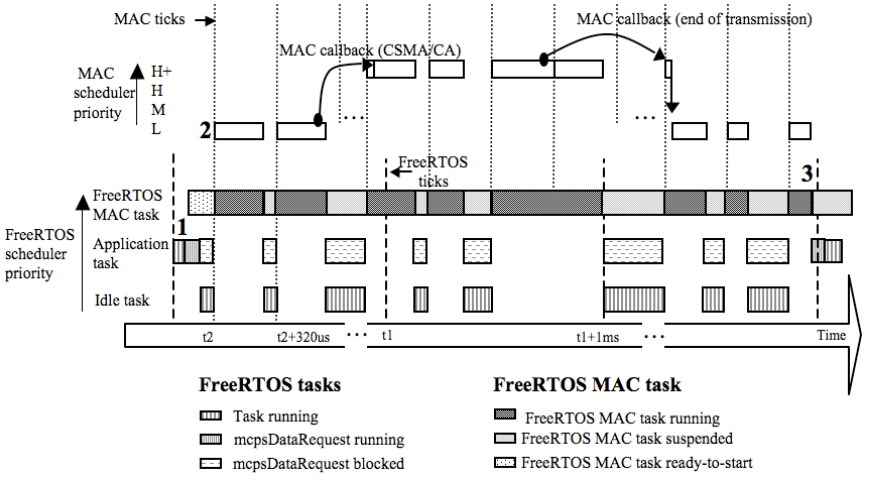

Fig. 5. Process scheduling for a packet transmission within the OS-based implementation

FreeRTOS *Idle* task runs. At the next MAC tick, the next state of *mtxScheduleTransmission* is executed and so on.

3. Later, when the transmission process is done, the FreeRTOS MAC task is suspended, the MAC tick is stopped and the application task unblocked. It will remain in this state until the application calls a new MAC primitive to initiate a service.

Synchronized distributed processing. Multiple tasks, some of which of periodic nature, are performed by sensor nodes in a network. It can be of interest to synchronize the execution of specific periodic tasks; synchronizing sensing tasks can improve data correlation, and can even be an unavoidable requirement when different signals have to be concurrently measured to deduce a global state. A synchronized periodic on/off switching of radio transceivers would also be beneficial for duty-cycling, by reducing the uncertainty period between the wake-up time of a transmitting node and a receiving node.

The method we devised to implement distributed task synchronization is based on synchronizing timer interrupt occurrences at different nodes to that of a single master [24]. The synchronized interrupts are then used to launch the execution of periodic tasks in a synchronized manner across the network. The method builds on the time synchronization and micro-kernel framework services and achieves task synchronization with high accuracy by specifically targeting the timer (micro-kernel timer) used by the micro-kernel for starting task execution and switching between tasks.

Our time synchronization implementation yielded an average absolute time synchronization error of 1 μs for nodes at one hop from the time master. This average error increases by less than 0.5 μs for every additional hop further from

the time master in a multihop scenario. The tests we conducted in a single-hop network of 5 nodes showed that we achieved accurate task synchronization over nodes, with an average absolute offset of 5 μs between the start of execution of the same task at the master and a second node. 90% of the registered 1000 offset instances were found to be smaller than 10 μs [24].

4.2 Memory Footprint

We measured the code size of the software framework on our CoolFlux DSP port, as shown in Table 2. The current version of the MAC software includes support for both beacon and non-beacon networks. Two buffers of length 128B for ongoing and incoming packets are used. Improvements can be realized to reduce the memory footprint. The first objective of the software implementation was to validate the concept, and less effort has been put into optimizing the coding.

Table 2. Measurements of FreeRTOS port to CoolFlux DSP

FreeRTOS code size	1757 words
(Words of 32 bits)	
FreeRTOS RAM usage	160 words
(Words of 24 bits)	23 words per task + stack
	26 words per queue + data
Hardware abstraction	9KB
MAC software code size	11KB
MAC software RAM usage	2.3KB

4.3 Micro-kernel Latency Overhead

The values presented in Table 3 compare the FreeRTOS context switch times on different microcontrollers, namely NXP CoolFlux DSP, TI MSP430 and CC2430's 8051. A context switch is the process of storing and restoring the state (context) of the processor such that multiple processes can share a single processor.

These timing measurements are very dependent on the clock frequency and the architecture of the module tested. They show that FreeRTOS (and more generally any micro-kernels) performance and overhead are very dependent on the platform. In order to quantify the overhead of the micro-kernel and other software services with respect to the *useful* processing, we measured on the NXP CoolFlux DSP the duration of a RF packet transmission, initiated by an application process, as shown in Table 4. The implementation and execution of the MAC state machine makes use of the FreeRTOS micro-kernel services and the hardware abstraction.

During transmission, FreeRTOS context switches happen at each MAC tick interrupt, every 320μs, and at each FreeRTOS tick interrupt, every 1ms. We conclude that with approximately four ticks per ms, a FreeRTOS context switch of 17μs represents an overhead of about 7% on the available processing power.

Table 3. FreeRTOS context switch duration for several architectures. (1) corresponds to a tick interrupt which does not wake-up any task, (2) is for a tick interrupt that wakes up one task and (3) refers to a task yield.

Processor	Tick (1)	Tick (2)	Yield (3)
CoolFlux (20MHz)	12.9μs	16.6μs	9.6μs
MSP430F449 (8MHz)	37μs	54.4μs	30.8μs
8051 (CC2430) (32MHz)	121μs	249μs	128μs

Table 4. Timing measurements on the CoolFlux DSP executing an 802.15.4 MAC within a FreeRTOS task (RF channel free)

Sending 1 bytes	2.6 (ms)
Sending 100 bytes	9.4 (ms)
Receiving 1 bytes	1.9 (ms)
Receiving 100 bytes	8.4 (ms)

4.4 Portability

The implementation of the software framework has been ported in three months to a TelosB-like node (TI MSP430 + CC2420) by a master student. The MSP430 port in itself only involved compiler adaptations, timer settings and adaptation of the lower layers of the hardware abstraction. The FreeRTOS port for the MSP430 was already available [2]. We also ported part of the software framework on the CC2430's 8051 microcontroller, but the latency related to FreeRTOS context switch (as shown in Table 3) introduced too much overhead to the system.

5 Conclusions

We proposed and evaluated a software framework dedicated to wireless sensor nodes. Both the architecture and services ease the development of applications, where power conservation with on-node processing is facilitated. We have provided software services, APIs, and contructs for temporal control over process execution and portability. The software framework has been ported to different microcontrollers, and the different modules have been validated by several applications. According to the requirements of application scenarios, a subset of requisite modules can be chosen, with the whole system providing a complete and efficient programming interface to the application developer.

Acknowledgments

The authors wish to thanks Albert Rietema for his work on the software implementation of the hardware abstraction and the MAC, and Nils Preusker and Tao Xu for their efforts on porting software components to the 8051 and the MSP430.

We would also like to thank Niek Lambert and Victor van Acht for their valuable feedback. This work is partially financed by the European Commission under the Framework 6 IST Project Wirelessly Accessible Sensor Populations (WASP) and is supported by Science Foundation Ireland under grant 07/CE/I1147.

References

1. Ouwerkerk, M., Pasveer, W.F., Engin, N.: SAND: a modular application development platform for miniature wireless sensors. In: Proc. of BSN 2006 International Workshop on Wearable and Implantable Body Sensor Networks, pp. 166–170 (2006)
2. FreeRTOS^{TM} Homepage, Richard Barry, `http://www.freertos.org/`
3. Texas Instruments CC2420 2.4 GHz IEEE 802.15.4 RF Transceiver Data Sheet, `http://focus.ti.com/docs/prod/folders/print/cc2420.html`
4. Maroti, M., Kusy, B., Simon, G., Ledeczi, A.: The Flooding Time Synchronization Protocol. In: Proceedings of the 2nd Int. Conf. On Embedded networked Sensor systems (SenSys) Baltimore (2004)
5. Willmann, R.D., Lanfermann, G., Saini, P., Timmermans, A., te Vrugt, J., Winter, S.: Home Stroke Rehabilitation for the Upper Limbs. In: Proc. of the 29th Annual International Conference of the IEEE EMBS Cite Internationale, Lyon, France (2007)
6. van Acht, V., Bongers, E., Lambert, N., Verberne, R.: Miniature Wireless Inertial Sensor for Measuring Human Motions. In: Proc. of the 29th Annual International Conference of the IEEE EMBS Cite Internationale, Lyon, France (2007)
7. Westerink, J.H.D.M., Ouwerkerk, M., Overbeek, T.J.M., Pasveer, W.F., de Ruyter, B. (eds.): Probing Experience - From Assessment of User Emotions and Behaviour to Development of Products. Philips Research Book Series, vol. 8 (2008)
8. Schoofs, A., Daymand, C., Sugar, R., Mueller, U., Lachenmann, A., Kamran, S.M., Gefflaut, A., Thiem, L., Schuster, M.: Testbed for IP-Based Herd Monitoring. In: The 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN (2009)
9. Adi Mallikarjuna, V.R., Phani Kumar, A.V.U., Janakiram, D., Ashok Kumar, G.: Operating Systems for Wireless Sensor Networks: A Survey, Technical Report (2007)
10. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D., Werner Weber, J.M.R., Aarts, E. (eds.): TinyOS: An Operating System for Sensor Networks, pp. 115–148. Springer, Heidelberg (2005)
11. McCartney, W.P., Sridhar, N.: Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In: Proceedings of SenSys 2006, pp. 167–180 (2006)
12. Duffy, C., Roedig, U., Herbert, J., Sreenan, C.J.: Adding Preemption to TinyOS. In: Proceedings of the The Fourth Workshop on Embedded Networked Sensors (EmNets 2007), Cork, Ireland (2007)
13. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: First IEEE Workshop on Embedded Networked Sensors (2004)
14. Chen, S.: Secure Real-time Services for Wireless Sensor Networks in Contiki (2007)
15. Klues, K., Hackmann, G., Chipara, O., Lu, C.: A Component-Based Architecture for Power-Efficient Media Access Control in Wireless Sensor Networks. In: ACM SenSys 2007, Sydney, Australia (2007)

16. Handziski, V., Polastre, J., Hauer, J.-H., Sharp, C., Wolisz, A., Culler, D.: Flexible hardware abstraction for wireless sensor networks. In: Proceedings of the Second European Workshop on Wireless Sensor Networks, EWSN (2005)
17. Fernando Friedrich, L., Stankovic, J., Humphrey, M., Marley, M., Haskins, J.: A survey of configurable component-based operating systems for embedded applications. IEEE Micro 21(31), 54–68 (2001)
18. IEEE 802.15.4 Standard-2003, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), IEEE-SA Standards Board (2003)
19. Freescale Semiconductors, 802.15.4 MAC PHY Software Reference Manual Rev. 1.6, IEEE Micro 22(6) (2008), `http://www.freescale.com/files/rf_if/doc/ref_manual/802154MPSRM.pdf`
20. Roeven, H., Coninx, J., Ade, M.: CoolFlux DSP - The embedded ultra low power C-programmable DSP core. In: Proceedings of the Int. Signal Processing Conf. (GSPx), Santa Clara (2004)
21. Ganeriwal, S., Kumar, R., Srivastava, M.B.: Timing-sync Protocol for Sensor Networks. In: Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, California (2003)
22. Elson, J., Girod, L., Estrin, D.: Fine-Grained Network Time Synchronization using Reference Broadcasts. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, Massachusetts (2002)
23. Aoun, M., Schoofs, A., van der Stok, P.: Efficient Time Synchronization for Wireless Sensor Networks in an Industrial Setting. In: Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems, SenSys (2008)
24. Aoun, M., Catalano, J., van der Stok, P.: Distributed Task Synchronization in Wireless Sensor Networks. In: Proceedings of the 6th European Conference on Wireless Sensor Networks (2009)
25. Andree, M., et al.: Core Hardware Abstraction and Programming Model, Deliverable D3.2, IST-034963, WASP (2008)
26. The Open Group Base Specifications Issue 6 (cited: 2008-04-01). IEEE Std 1003.1-2001, The IEEE and The Open Group, `http://www.unix.org/online.html`
27. Aldea Rivas, M., Gonzalez Harbour, M.: 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, Porto, Portugal (2003)