



Research Repository UCD

Title	Portability in MAC protocol and transceiver software implementations for LR-WPAN platforms
Authors(s)	Schoofs, Anthony, Stanley-Marbell, Phillip
Publication date	2011-04-10
Publication information	Schoofs, Anthony, and Phillip Stanley-Marbell. "Portability in MAC Protocol and Transceiver Software Implementations for LR-WPAN Platforms." Wiley, April 10, 2011.
Publisher	Wiley
Item record/more information	http://hdl.handle.net/10197/2523

Downloaded 2025-12-04 22:59:48

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)



© Some rights reserved. For more information

Portability in MAC protocol and transceiver software implementations for LR-WPAN platforms



Anthony Schoofs^{*,†}, Phillip Stanley-Marbell[§]

SUMMARY

In a variety of emerging networked computing system domains over the years, there have been bursts of activity on new medium access control (MAC) protocols, as new communication transceiver technologies with greater data-movement performance or lower power dissipation have been introduced. To enable implementations flexible to evolving standards and improving application-domain insight, such MAC protocols are typically initially implemented in software, and interface between applications or system software, typically executing on an embedded processor or microcontroller, and the evolving radio transceiver hardware.

Many challenges exist in implementing MAC protocols across evolving or competing transceiver hardware implementations and processor architectures. Some of these challenges are peculiar to the requirements of MAC protocols, and others are a result of the plethora of system and processor architectures in the embedded systems domain. This article studies the challenges facing software implementations of MAC protocols running on embedded microcontrollers, and interfacing with radio transceiver hardware. Experience with an implementation of the IEEE 802.15.4 MAC across three hardware platforms with different processor, system, and systems software architectures is presented, focusing on *implementation approach* and *interfaces*. Pitfalls are pointed out, and guidelines are provided for ensuring that new MAC implementations are easily portable across processor architectures and transceiver hardware.

KEY WORDS: Portability, medium access control (MAC) protocol, device driver, hardware abstraction, IEEE 802.15.4, LR-WPAN, wireless sensor networks.

*Correspondence to: Anthony Schoofs, CLARITY: Centre for Sensor Web Technologies, University College Dublin, Dublin, Ireland

[†]E-mail: anthony.schoofs@ucd.ie

[§]IBM Research—Zürich, Switzerland.

[§]E-mail: pst@zurich.ibm.com

Contract/grant sponsor: European Union IST program and Science Foundation Ireland; contract/grant number: WASP IST-034963 and SFI 07/CE/I1147

1. Introduction

Networks of communicating resource-constrained systems are a rapidly evolving domain. As technology advances, research interest and commercial applications drive the creation of new system architectures and hardware implementations. An important category of networked embedded computing systems is that of low-data-rate wireless personal area networks (LR-WPANs) or wireless sensor networks. LR-WPANs need to efficiently make use of the wireless communication spectrum; as communication operations in these systems often dominate the system's power dissipation, the achievement of energy- and time-efficient communication is a primary concern. Key to achieving efficient communication is the acquisition of the communication channel, and sharing it among multiple competing transmitters; this function is achieved via the use of *medium access control (MAC)* protocols. Due to the rapid evolution of hardware and software platforms, MACs are often implemented in software executing on a *control processor* controlling the actual *radio frequency (RF) transceiver*. This article addresses the challenges in implementing MACs, such that they are *portable* across radio transceivers (which they control), across system software platforms, and across processor architectures over which they execute. The work presented is based on experience with an IEEE 802.15.4 MAC implemented across a variety of system architectures, processor architectures, and transceivers.

1.1. Medium access control (MAC) protocols

MAC protocols serve the primary role of reducing collisions—the occurrence of multiple simultaneous transmissions arriving at a receiver. In most communication interface architectures, such overlapping incoming communications lead to the inability to correctly receive all (or at least, one) of the multiple overlapping incoming transmissions, as illustrated in Figure 1(a). In addition to avoiding collisions, MACs enable a system to adapt to varying on-node and environmental conditions, in order to maximize throughput and minimize delay. Examples of on-node conditions include available energy resources, while examples of environmental conditions include changes in RF noise over time and location.

In a wired communication medium, it may be possible to detect the occurrence of a collision by sensing the current along the communication conductor, as occurs in IEEE 802.3 (Ethernet) transceivers. In wireless communication channels however, it is generally not possible to achieve collision detection on the sender side, for two primary reasons. First, a receiver may be located such that it is reachable by transmissions from two or more transmitters, which are mutually unreachable by wireless transmissions; this is a form of the *hidden terminal problem*. The second less fundamental reason is due to the architecture of most low-power short-range wireless transceivers: most include a single antenna, and incorporate a radio frequency (RF) switch to select between connection of the receive path *low noise amplifier (LNA)* versus the transmit path *power amplifier (PA)* (Figure 2). As a result, it is not possible to perform full-duplex communications in such architectures: it is not possible to sense the channel *during* transmissions.

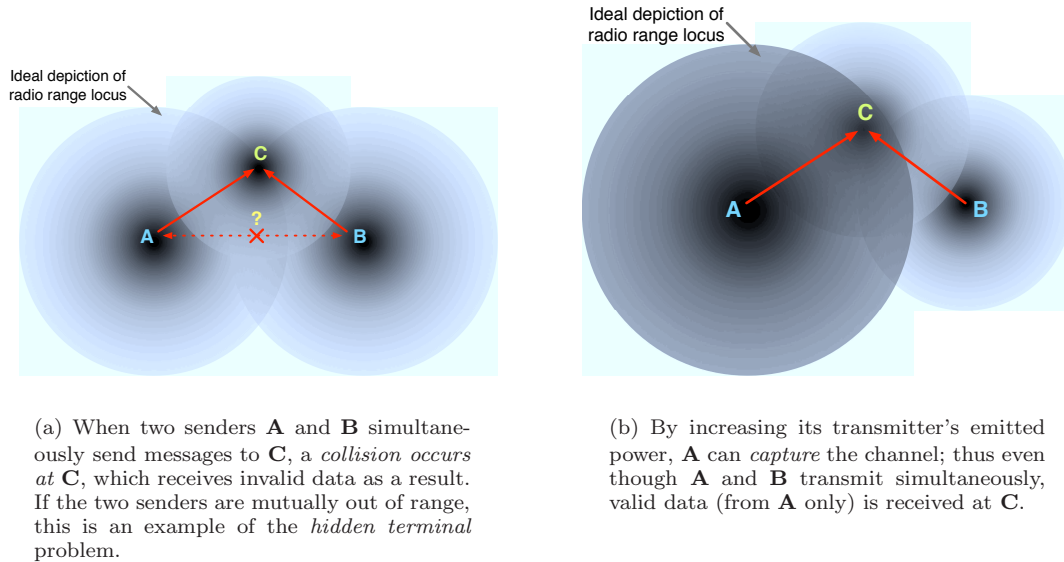


Figure 1. Illustration of the ideas behind (a) *collisions* and (b) *channel capture*. The primary role of transceiver software, and MAC protocols in LR-WPAN platforms, is to proactively reduce the occurrence of collisions, via arbitration of the communication medium, and transceiver power control.

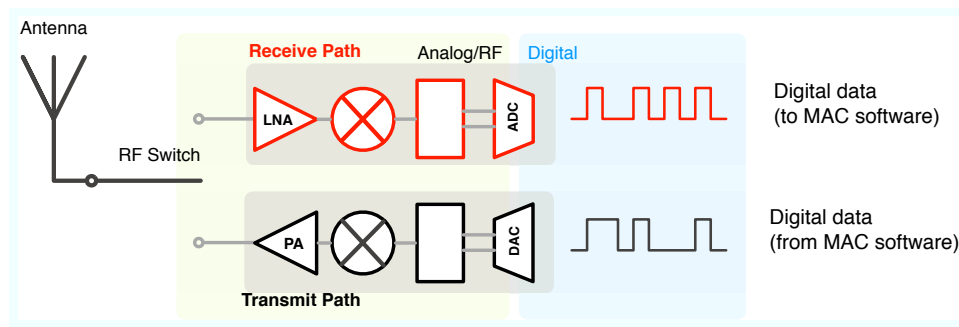


Figure 2. Typical transceiver RF signal path architecture, highlighting the half-duplex nature of most commercial LR-WPAN transceivers, which precludes the monitoring of the channel at the same time as an ongoing transmission on the same device. At a given moment, only one of the two signal paths is connected to the antenna via the RF switch.

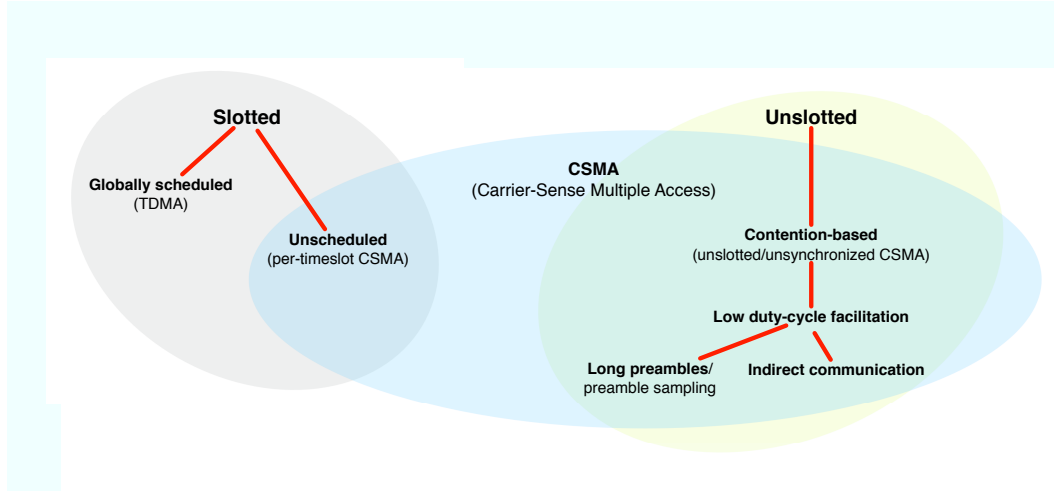


Figure 3. A classification of the typical types of medium access control protocols employed in LR-WPANs. The different classes of protocols require different implementation facilities, e.g., low-level message timestamping, hardware clock drift compensation or software-based time synchronization.

In order to proactively prevent collisions, a variety of mechanisms are implemented in MAC protocols, classifying MACs into the main groups of *random access MACs* with uncoordinated access to the wireless medium, and *slotted MACs* with access to the wireless channel occurring in (locally) coordinated time slots (Figure 3). These classifications have direct impact on implementation and portability—slotted MACs require more interaction with timers, and in the absence of appropriate primitives in system software, protocol implementors often resort to non-portable platform-specific solutions. This variety of MAC types is however necessitated by the range of environments in which LR-WPANs are deployed. These deployment environments range from remote natural habitats to densely populated urban areas, and each pose a different set of network communication constraints.

Evolving hardware capabilities lead to the evolution of the algorithms for channel access that MAC protocols embody. For example, in recent years, the idea of *channel capture* has re-emerged as a phenomenon of interest in LR-WPAN deployments, due to the availability of transceivers that provide hardware facilities for taking advantage of it. Similarly, the introduction of silicon implementations with support for full-duplex RF signal paths may lead to a revisiting of the assumption that collisions (of the second form in the foregoing description) cannot be detected at the sender, and may further lead to new proposals for more efficient medium access control protocols.

1.2. Contributions and outline

The rapid co-evolution of hardware, algorithms and software makes it necessary to consider software implementations of MAC protocols, typically executing in a small control processor or microcontroller attached to an RF transceiver, or embedded in the transceiver itself. This article addresses the challenges posed by the goal of implementing MAC protocols in such a manner that they may be employed with different control processor / microcontroller architectures, as well as with different wireless transceiver architectures. These challenges go beyond judicious software engineering practices, and are partly a result of the peculiar challenges posed by the requirements of MAC protocol implementations, and the restrictions of typical transceiver hardware and embedded processors. The specific challenges posed by such *portable MACs* include:

- Achieving portability and appropriate abstractions without compromising performance.
- Accommodating the diverse set of transceiver capabilities.
- Dealing with the variety of communication interfaces between transceivers and the control processor, e.g., via universal asynchronous receiver transmitter (UART), or serial peripheral interface (SPI) links.
- Providing a *bottom interface abstraction* or API, from MAC implementations to transceivers and microcontrollers (Figure 4).
- Providing a *top interface abstraction* or API between applications and MAC implementations (Figure 4).
- Providing a *system interface abstraction* between system software and MAC implementations (Figure 4).
- Defining a MAC implementation strategy, e.g., a choice of the use of co-routines, state machines or communicating processes, as well as addressing issues such as management of *periodic timers* and single *timeout events*.

Following a discussion of related research in Section 2, Section 3 discusses the state of the art and challenges to portability posed by radio transceiver hardware features, control processor capabilities, system software and compilation tools. Section 4 presents detailed accounts in porting an implementation of the IEEE 802.15.4 MAC across three hardware platforms with different processor, system, and systems software architectures, focusing on implementation approach, and the bottom and system interfaces. It is followed in Section 5 by insights and practical guidelines for future MAC implementations for LR-WPAN (and similar) platforms. Section 6 concludes the article with a summary and discussion of possible challenges for future systems.

2. Related Research

In the literature on low-power embedded sensor systems such as wireless sensor networks, medium access control protocols are one of the more heavily studied areas, evident in the large number of proposals for MAC protocols (e.g., [11, 15, 29, 30, 44]); despite this large interest therein however, many proposed MAC protocols have only been implemented in simulation.

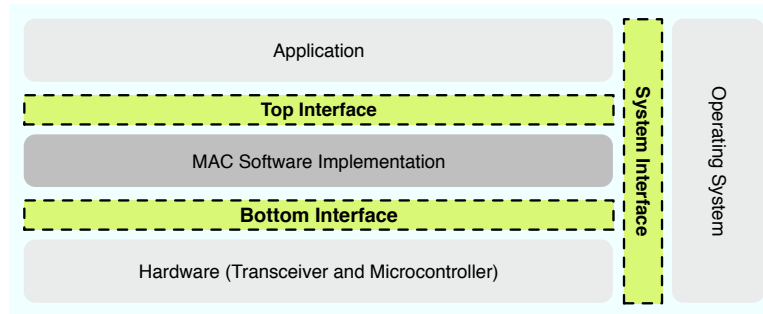


Figure 4. The interfaces between the MAC implementation and applications, hardware, and the system software (operating system or other dedicated embedded software), will be referred to in the remainder of this article as the *top*, *bottom* and *system software interfaces*.

There are many challenges in real-world implementations of MAC protocols, as the implementations must deal with the idiosyncrasies of hardware devices, sometimes in the presence of hardware bugs. Despite these challenges, there has been little coverage in the research literature of the specifics of addressing such implementation hurdles in the context of low-data-rate wireless personal area networks (LR-WPANs), e.g. [34]. One possible reason for the apparent lack of interest in *implementations* of MACs that may be portable across embedded operating system platforms, transceivers, and processor architectures, might be the misconception that the abstractions that must be introduced to enable such portability necessarily come with a significant overhead. Abstraction, however, does not necessarily imply large overheads and poor performance—if appropriately designed, interfaces can be low overhead [24], and may even enable added efficiency, by providing additional control of the protocol behavior to applications [29].

2.1. Better portability through abstraction

The idea of abstraction pervades all aspects of computer science and engineering, and is particularly prevalent in computing systems research and operating systems. All contemporary operating systems employ some form of abstraction. While such abstractions may be traced back to early research systems such as Multics [26], the Unix operating system [31], and its progeny, it is instructive to concentrate on four techniques targeted specifically at abstracting hardware resources—(1) application programming interfaces (APIs) for specific layers in network protocol stacks of resource-constrained embedded systems, such as ULLA [36]; (2) the 4.4 BSD device driver interface; (3) the Plan 9 and Inferno operating system device interfaces; (4) driver, protocol and networking stack specification languages such as Devil [23], NDL [7], Zebu [6], Prolac [18], and Click [19]. While occurring in different topic areas of the computing systems milieu, the insights from these systems are relevant to LR-WPANs.

The *Unified Link-Layer API (ULLA)* [36] provides a common interface to a variety of link layers (e.g., 802.11, Bluetooth, 802.3). The premise of ULLA is that there is a need to isolate application programmers from the peculiarities of different link layer implementations. ULLA provides a “thin” version of what one may refer to as the “top” interface to a link layer implementation, further split within ULLA into a *link provider* interface (in essence the lower half of the “top” interface) and a *link user* interface (the upper half of the “top” interface). ULLA is however neither concerned with the portability of the actual link layer implementation across radio transceiver hardware, nor with portability of the link layer implementation across operating system platforms which may provide different facilities to the link layer implementation. A prototype ULLA implementation [36] occupies a memory footprint of approximately 200KB, an order of magnitude more than the total memory available on the platforms of interest in LR-WPANs.

The form of isolation targeted in ULLA has been present for many decades in the Plan 9 operating system [28], a research operating system developed by the creators of Unix as its successor, and in progeny of Plan 9, such as the Inferno operating system [43]. Unlike in Unix variants (e.g., 4.4 BSD [22]), where I/O may occur on distinct object types of *files*, *unidirectional pipes*, or *sockets*, Plan 9 and Inferno use a single abstraction—*files*—for all types of static data, dynamic data, and communication paths. In both Plan 9 and Inferno, the various layers in the network protocol stack are accessed using *an abstraction of a filesystem*, for both data exchange and control. This contrasts Unix implementations where there exist filesystem abstractions (*device special files*) for, e.g., disk devices, but control needs to be performed through *ioctl* system calls. Similarly, while the low-level interface to devices such as disks are typically through device-special *files*, the low-level interfaces to network devices are typically through *raw sockets*. The typical network resource in Plan 9 is implemented with three file abstractions: (1) a *clone* file which can be read to create a new network connection; (2) a *data* file abstraction for sending and receiving data (e.g., link layer data); (3) a *ctl* file abstraction for controlling the configuration of a connection. The file abstraction provides a consistent and well understood interface for sending data down a network layer (writes to *data*), waiting for incoming data (blocking reads from *data*) and changing of configuration parameter values (writes to *ctl*); these abstractions and their use are identical regardless of whether the network facility in question is one of many link layers (e.g., IEEE 802.3 Ethernet or 802.11 WLAN) or a network layer protocol (e.g., TCP or UDP). The Plan 9 and Inferno network stacks provide examples of how a consistent interface can ease the portability of network software.

An alternative approach to achieving portability of low-level software is through the use of an intermediate specification language. One such example is Devil [23]. Devil is an interface definition language (IDL) for specifying device drivers—the low-level software that interacts with specific hardware devices, such as the radio transceivers that are of interest in this article. Device drivers typically form the “bottom” portion of medium access control protocols. Like Devil, NDL [7] is another domain-specific language for specifying device drivers. The benefits of Devil, NDL, and their associated tools, is the expedient creation of the software that interfaces with hardware devices. They are not, however, designed with the implementation of protocols (which may be driving these low-level interfaces) in mind. Tools such as Prolac [18] and Zebu [6] on the other hand, are not concerned with direct interface to hardware, but rather, are concerned with providing facilities for implementing protocols.

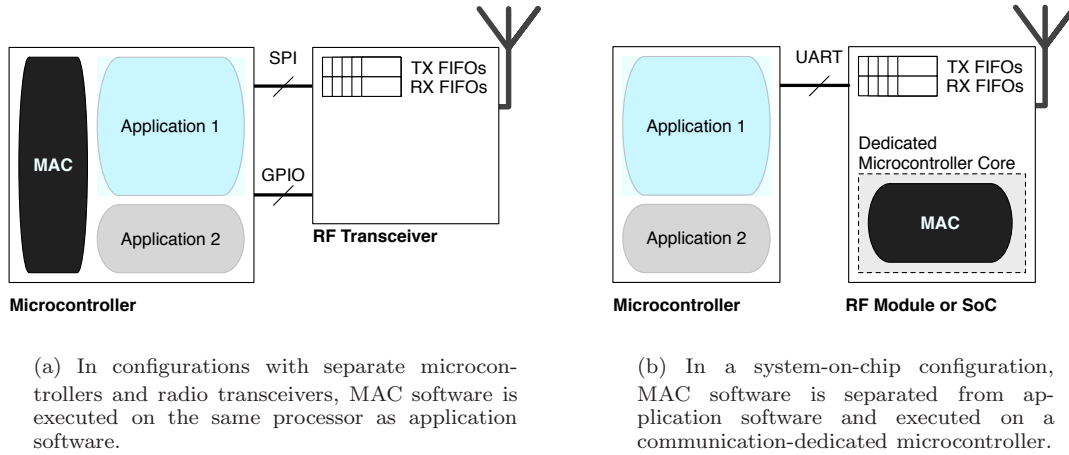


Figure 5. Illustration of the typical hardware configurations for LR-WPAN platforms, and their effect on the IEEE 802.15.4 MAC execution.

The structural organization of network stack interfaces and the tools for implementing portable device drivers and protocols provide many important lessons for implementing portable LR-WPAN MACs and transceiver interface software. It is however also important, in the context of the topic of this article, to consider the peculiar constraints imposed by the IEEE 802.15.4 standard on the portability of MAC implementations.

2.2. IEEE 802.15.4 MAC implementations

Approaches to IEEE 802.15.4 MAC implementations for LR-WPANs platforms differ in where and how the MAC protocol is executed. LR-WPANs platforms [9, 13, 25] (often also referred to as *sensor nodes*), are typically designed with a separate microcontroller and radio transceiver (Figure 5(a)). Most radio transceivers do not have all functionality required for IEEE 802.15.4-compliant operation implemented directly in hardware, as shown in Table I. MAC protocol mechanisms (e.g., *carrier sense multiple access with collision avoidance (CSMA/CA)* for accessing the RF medium) are thus often implemented in software on the platform's microcontroller.

Some radio transceivers, such as the TI CC2430 [38] include an integrated microcontroller for executing protocol-related functions. Applications can run either on this integrated processor core, or on an external application-dedicated controller. In the latter case, the separate microcontroller communicates with the transceiver via a serial bus (e.g., SPI or UART) (Figure 5(b)). Radio transceivers may also implement portions of the IEEE 802.15.4 protocol in hardware. For example, the Atmel AT86RF230 transceiver [3] implements CSMA/CA as

a hardware-accelerated feature. However, when taking advantage of hardware-acceleration of protocol subsets, porting to new platforms may require more effort if appropriate abstraction layers are not put in place.

Most IEEE 802.15.4 MAC software implementations are targeted at microcontrollers allowing nesting and prioritization of interrupts. For example Freescale provides such an 802.15.4 MAC implementation, which is independent of an operating system [12]. Similarly, Texas Instruments also provides an interrupt-based implementation of the 802.15.4 MAC layer, for its CC2420 transceiver [37]. When migrating these implementations to hardware platforms with limited interrupt handling capabilities, rewrites of (possibly large) portions of the codebase will be necessary.

3. Challenges in Portable IEEE 802.15.4 MAC Implementations

The four primary challenges to implementing medium access control protocols that are portable across radio transceivers and system platforms, are: (1) the diversity in radio transceiver interfaces and feature sets; (2) the variety of system-level hardware features such as processor peripherals in low-power control processors; (3) the variety of operating system services such as memory allocation and synchronization primitives (the *system software* interface of the MAC); (4) the diversity of instruction set architecture features and compiler support.

3.1. The diversity of hardware features in 2.4 GHz WPAN transceivers

Radio frequency transmitters/receivers (transceivers) form the interface between digital data communication systems and the analog radio frequency modulation, transmission, demodulation during reception, and final receipt of digital data in the destination network stack. Properties of interest in transceiver performance include the receiver sensitivity and output/emitted power (which together determine the *link budget*—in essence how much noise and interference can be withstood between a transmitter and its destination)—, power dissipation at various emitter transmitter power levels, and idle power consumption. There exist no common interfaces for LR-WPAN transceivers as there were in the early nineties for Ethernet transceivers, in the form of the NE1000/NE2000 de-facto interface standards. This lack of a common interface presented to MAC implementations is reflected in the frequent need for modification of MAC protocol implementations during porting.

An important criterion for comparing transceivers is the facilities they provide to ease MAC layer processing, ranging from hardware support for frame checksums and flexible ingress and egress frame buffers, to automatic acknowledgments of received frames, and interfaces through which MAC implementations may query or modify transceiver state. Table I provides an overview of the facilities provided by several contemporary LR-WPAN transceivers. A MAC implementation that must interface to this collection of transceivers must therefore cope with variations in the interface (e.g., SPI, I2C, UART, etc.), cope with providing software implementations for hardware accelerated features on some platforms (e.g., encryption, CSMA/CA), and must elide dependencies on hardware receive signal strength

indication (RSSI) readouts on platforms on which it is unavailable. Such an implementation must also be able to map its power saving strategies to available hardware power saving modes.

3.2. The diversity of low-power control processors and their peripherals

Embedded system applications are typically cost-sensitive, due to the fact that the embedded computation and control is typically seen by the end-user as an auxiliary function, if even recognized at all. This is in contrast to traditional desktop and server application domains, where the “computer” is well recognized as a goal in itself, rather than a means to an end. As a result, embedded system applications are highly integrated and constrained, to minimize cost. They usually combine computation (arithmetic and logic units), with memories (typically on-chip static random access memory (SRAM)), and peripherals such as communication interfaces, timers, and analog-to-digital or digital-to-analog converters. The lower end of the spectrum of such devices typically have internal processor bus widths of between 4 and 16 bits, with only on-chip memory, no interfaces to external RAM, and are usually referred to as *microcontrollers*. The higher end of the spectrum are typically 32- or even 64-bit systems, with higher clock rates, caches, and interfaces to external memories such as dynamic random access memories (DRAMs). The latter are usually referred to as *embedded processors*. In recent years, when containing high degrees of integration of peripherals that have traditionally been contained in separate integrated circuits, they are increasingly referred to as *systems-on-chip* (SoCs).

MAC implementations and transceiver interface software for LR-WPAN platforms are typically implemented on 8- or 16-bit microcontrollers. The needs of the implementations depend on the type of MAC protocol (e.g., slotted versus unslotted, Figure 3), and the implementation constraints are determined by the facilities provided by the microcontroller on which they are implemented. In particular, it can be observed from Figure 3 that an important aspect of MAC implementations is timing (for slotted protocols, as well as to determine low-power sleep schedules). Other constraints include the microcontroller’s interrupt mechanism, available memory and memory access restrictions, and hardware acceleration of operations such as multiplication and *cyclic redundancy check (CRC)* computation. Table II illustrates the range of these facilities for a selection of contemporary microcontrollers used in LR-WPAN platforms. The following details how this diversity of facilities affects both MAC implementation and portability.

3.2.1. Timer Engines

Most microcontrollers provide facilities for managing time-based events, such as initiating periodic actions, or timing monitored signals, through what are usually called *timer engines* or *timer peripherals*. Timer peripherals are of particular importance in slotted MAC protocol implementations (see Figure 3), as they enable the synchronization of protocol operations to a local or global time base, as well as the recording of the time of occurrence of protocol events (e.g., incoming packets).

Timer peripherals typically comprise two components. A *counter*, typically 8 or 16 bits, may be configured by software to count *up* or *down* in a variety of schemes. The counters are usually

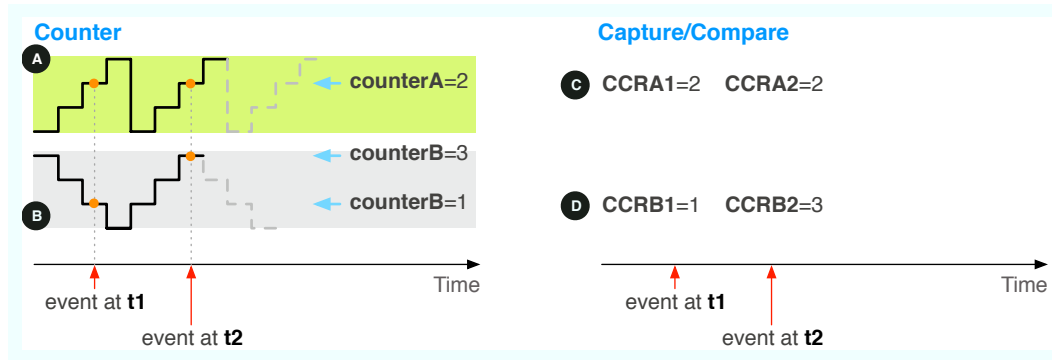


Figure 6. Illustration (using simple 2-bit counters) of how *counters* and *capture/compare* registers are used together in a system to monitor the occurrence of events in time, or to trigger periodic actions: (A) an “up-up” counter and (B) a “down-up” counter. Events occurring at times t_1 and t_2 may be captured in registers (**CCRA1** and **CCRA2** for the “up-up” counter). The configuration of **counterA**’s compare register to a value of 2 triggers periodic actions every 4 clock cycles.

driven either by an external clock source, a crystal-driven *phase-locked loop (PLL)*, or an on-chip R-C oscillator. Counters are typically used in conjunction with a second component, *capture/compare registers*. These are used to hold a snapshot (i.e., *capture*) of the counter at the point of occurrence of an event of interest. Conversely, the timer peripheral may be configured to generate an interrupt when the value of one of its counters matches a previously set value in a capture/compare register. A single timer counter, in conjunction with multiple capture/compare registers, may thus be used to generate multiple periodic actions, or to monitor the occurrence of multiple events. Capture/compare registers thus permit capturing the time of occurrence of events, without the need for potentially costly interrupt-triggered software timer snapshots. Figure 6 illustrates.

Due to the role event timing plays in most MAC protocols, the diversity of timer hardware is an important factor affecting the portability of MAC implementations across platforms. As examples of these facilities in a sampling of processors used in LR-WPAN platforms, the NXP CoolFlux DSP [32] provides two 16-bit timers, which can be configured to count down to zero, interrupting the system, but provides no timer compare registers. In contrast to the CoolFlux, the TI MSP430 [39] and the Atmel ATmega128 [8] microcontrollers provide multiple timers, and multiple capture/compare registers. These differences in the timer architectures have important implications for MAC implementations on the CoolFlux versus, say, on the MSP430. For example, on the CoolFlux, time-stamping has to be handled in software, where interrupts generated by events of interest initiate a software snapshot of the timer counter. Overhead, latency and uncertainty in measuring the time of occurrence of events is thus introduced.

As one example, the use of timer capture registers was essential for the implementation of an FTSP-like time synchronization protocol on the ATmega128 [2]. In that implementation, time captures were triggered when a change of the logic level (i.e., an event) occurred on the *input*

capture pin of the ATmega128, connected to the *start of frame delimiter (SFD)* output pin of a CC2420 transceiver. This would then cause the writing of the ATmega128's 16-bit counter value into its *input capture register*, providing a timer snapshot for the received packet. These challenges, and possible solutions, are addressed in Section 4.

3.2.2. Interrupt management units

Many computing systems are, in some form, event-driven. This event-driven nature is particularly pronounced in microsensor platforms, where the primary activity of a system is to react to the occurrence of some temporal, communication, or sensing event. In virtually all systems, the change of normal program control flow in response to such events is through an *interrupt mechanism*, in which program execution (or idling) is halted, and the processor jumps to a pre-determined address to begin executing a new stream of instructions. The ability to return to its previous execution flow is often aided through a combination of hardware and software facilities to save some execution state, and possibly, to prevent the occurrence of further interrupts. If a processor can further take new interrupts during interrupt processing, it is said to be capable of taking *nested interrupts*.

The capability of an architecture to support the nesting of interrupts prevents the loss and delay of high-priority events, but affects the complexity of MAC implementations. In order to protect the execution of a critical piece of code that should not be interrupted (e.g., the use of a shared peripheral that must be accessed by only one process at a time), further interrupts must be appropriately masked. On architectures that do not support nested interrupts, realizing interrupt-driven software executions has some limitations. Higher-priority events are not capable of interrupting an executing interrupt service routine (ISR), forcing interrupt routines to be short in terms of instructions. This constraint is reflected in MAC implementation constructs, as the execution of MAC functions cannot be executed inside periodic ISRs.

When interrupts are *prioritized*, higher-priority interrupts are serviced first. The support for interrupt prioritization differs across microcontroller architectures. For instance, the ARM *generic interrupt controller (GIC)* architecture supports taking higher-priority pending interrupts before an active interrupt completes. On other platforms, prioritization may either not be possible, or may need to be emulated in software. Such is the case, for instance, on the CoolFlux DSP, which is discussed further in the study presented in Section 4. On the CoolFlux, ISRs are separated into three interrupt vectors. When an interrupt is generated, the processor begins execution of the interrupt handler at the appropriate interrupt vector, and interrupt status bits are checked sequentially to identify the source of the interrupt amongst the registered peripherals. Prioritization is realized by ordering the reading of status bits according to the priority given to each peripheral interrupt. Differences in interrupt prioritization support may, undesirably, be reflected in MAC implementations. If such architecture-dependent constructs see their way into a MAC implementation, the processing of high-priority MAC events might require significant changes as the implementation is ported across platforms.

3.3. The diversity of operating system services

System software environments (e.g., operating systems,) differ across platforms, even within the domain of LR-WPAN applications [4, 5, 10, 20]. Among other things, they vary in the system calls through which they abstract hardware and temporal resources. From the viewpoint of portable MAC implementations, the primary differences of interest are the interfaces to timer facilities, memory management, interrupt handling, interprocess communication, and hardware peripheral access.

3.3.1. Timing control for MAC execution

MAC protocols are often implemented as state machines, where each state represents a different phase in the communication protocol. For example, an *idle state* in a MAC protocol implementation may correspond to waiting for the arrival of a physical layer packet. Another state in a typical MAC handles the transmission of data, and this state often has stringent timing requirements. TDMA protocols pre-allocate and assign time slots to individual communicating entities (nodes) for sending packets, whereas CSMA protocols, broadly speaking, make the nodes compete for channel access. Both methods require control over timing, either for synchronizing nodes to a common notion of time, or for rescheduling carrier sensing in case of a busy channel[†]. Differences between timer engines available on different platforms, as shown in Section 3.2.1, together with the absence of OS timing control and peripheral hardware abstraction raise the possibility that MAC implementers might make the timing control of their implementations hardware-specific, via direct access to low-level timer registers and close coupling to the underlying timer engine.

3.3.2. Memory management for packet handling

Communication buffers are blocks of memory allocated to hold the contents of a frame for processing. Every communication stack must have an efficient method of buffer management, as it is one of the main factors that affect its memory usage and timing performance. One goal of a buffer management system is to minimize copying of data. Buffers can be implemented with either dynamic memory allocation, or, alternatively, via static memory allocation, with a pool of buffers created at initialization. Often, an initial static pool of buffers is created and further creation of buffers is possible via dynamic allocation of memory. Buffers allocated to contain received packets are later passed to lower/upper layers for further processing before being released to the pool for further use, or deleted to free memory.

Tradeoffs exist between the choices of static versus dynamic allocation of buffers. Dynamic allocation may lead to memory fragmentation, and is often not even supported by typical operating systems and runtimes for deeply-embedded systems. On the other hand, static allocation may be inflexible, if the sizes of buffers required may vary significantly at runtime.

[†]Slotted CSMA/CA repetitively delays a carrier sense by a random number of time slots in order to give each node competing for the channel the same chance of accessing it.

The choice of memory management scheme is thus influenced by both portability and performance constraints. The case study described in Section 4 opts for the use of static allocation in the case of an IEEE 802.15.4 MAC implementation, as the required sizes of buffers are small, and are known beforehand.

3.3.3. *Interrupt control for handling MAC events*

Many MAC protocol implementations use interrupts (generated by hardware timer engines), to drive the MAC state machine. In parallel, facilities of the timer engines are used for time-stamping packets during transmission and reception events, as well as for notifying the occurrence of events such as the reception of a packet. The interrupts are typically used to trigger code execution, e.g., of time synchronization algorithms based on timestamps made on packet reception. These pieces of code are usually small in order to return as soon as possible to a state where new events can be detected.

Typical implementations for executing MAC code on an interrupt event place code in the respective interrupt service routine. Confining MAC code outside of interrupt service routines to avoid code dependencies and programming intricacies can be realized via a subscription mechanism provided by the system software. In an interrupt subscription architecture, the MAC, like any other process, posts a subscription to have an appropriate subroutine execute for each type of interrupt. When an interrupt fires, the ISR checks the list of subscribers and executes the registered pieces of code.

3.3.4. *Inter-process communication*

Processes within a MAC protocol implementation, running over an operating system, often need to interact with other processes internal or external to the MAC, either to exchange data, or take action on a shared state variable. Communication between a MAC and the rest of a network stack (e.g., upper network layers, applications, or even a time-synchronization protocol) requires coordination. Such interaction could be made via global variables (guarded by mutual exclusion locks), or by other structures such as semaphores.

Processes that need to take action based on the value of a system state variable or a blocking semaphore protecting access to a peripheral, will block code execution until the process that holds the semaphore releases it. Queues may also be used to transfer data between independent processes. When using queues, data or events are stored by one process and held to be processed later, possibly by another process. A benefit of using queues is the decoupling between processes it enables, as a process placing an item into a queue does not need to know which process will retrieve it. Support for queues and semaphores is however often not provided on lightweight software platforms. Thus, in designing for portability, it may be necessary to implement these facilities in a portable manner, as part of the MAC implementation.

3.3.5. Interfaces to hardware peripherals

MAC protocols must control radio transceiver hardware, in order to execute MAC-related functions, e.g., sensing a communication channel before initiating packet transmission. MAC protocols may also need to configure transceiver properties, such as the (emitted) transmission power of the transceiver, by configuring the transceiver's on-chip registers. Switching on and off the radio and enabling auto-acknowledgments are other examples. This software control involves two steps: generating the correct configuration command to send to the transceiver, and handling the transmission of the bytes constituting this command, over a communication link, from the microcontroller to the transceiver integrated circuit. The use of a *radio driver* may facilitate mapping a MAC protocol's high-level commands to a transceiver-specific low-level sequence of bytes that will enable the desired configuration or operation. Such a separation of a MAC implementation into self-contained *MAC protocol processing* and a separate *radio driver* for interfacing to the radio transceiver hardware is an important step towards achieving portability.

A variety of communication interfaces may be used between transceivers and a control processor. Possible interfaces include *universal asynchronous receiver transmitter (UART)* or *serial peripheral interface (SPI)* links. These different interfaces may each be used in a variety of modes, e.g., by polling, or using interrupts to signal completion of byte-level transactions, or by employing DMA block-level transfers. Each of these options yields different performance versus code portability tradeoffs.

When transferring bytes by polling, software continuously samples the status of the communication interface to the transceiver, waiting until the interface becomes free; slow interfaces such as UARTs make this method inefficient. Interrupt-driven transfer of bytes, in contrast to polling, returns control to the executing program while bytes are transferred on the communication interface; the processor is interrupted when each transfer completes. In the case of DMA transfers, it is often possible to transfer data from a large block of memory (e.g., several hundred bytes) autonomously. Since the setup of interrupt-driven transfers may involve several steps, its choice (instead of polling) will depend on the relative overheads. With these observations in mind, a polling peripheral interface might be more appealing in some situations.

3.4. The diversity of instruction set architectural features and compiler support

Dependent on the microcontroller hardware architecture, memory may either be *byte-accessible* or *word-accessible*. For example, the compiler tools for the 24-bit word-accessible CoolFlux DSP [32] implements `char`, `short` and `int` C language types, all using a 24-bit machine word. On the other hand, 8-bit microcontrollers with byte-accessible memories, such as the ATmega128, implement the `char`, `short` and `int` C language types with 1, 2 and 2 bytes respectively. In designing MAC implementations to be portable across a variety of processor architectures therefore, care must be taken to avoid dependencies on machine word size. Furthermore, it may even be wise to include word-size-independent auxiliary routines for byte packing/unpacking as part of the portable implementation.

4. Case Study: Experience Porting an IEEE 802.15.4 MAC Implementation Across Multiple Platforms

The IEEE 802.15.4 standard [14] defines a physical layer (PHY) and a MAC layer for low-bit-rate and low-power wireless communication. The MAC layer provides two services: the MAC data service and the MAC management service. The data service enables the transmission and reception of MAC protocol data units (MPDUs) across the PHY layer. The features of the MAC management service include *association*, *disassociation*, *network formation*, and *scanning* of existing networks. Transmission of data or command frames relies on a *carrier sense multiple access with collision avoidance (CSMA/CA)* mechanism to schedule communication. The standard introduces a *superframe* structure, which defines a division of time into slots, where nodes compete for channel access in *contention access periods (CAPs)*; alternatively, *guaranteed time slots* may be reserved for direct communication without CSMA/CA.

Beacons are used in this *slotted* version of CSMA/CA to define the CAP. Conversely, if beacons are not being used, or if a beacon could not be located, the MAC sub-layer transmits using the *unslotted* version of the CSMA/CA algorithm. In both cases, the slots, also called *backoff slots*, have a duration of 20 *symbols*, with a symbol equal to $16\mu\text{s}$ [14]. In slotted CSMA/CA, the MAC sub-layer ensures that the PHY commences all of its transmissions on the boundary of a backoff slot. In unslotted CSMA/CA, the backoff slots of a given node are not related in time to the backoff slots of other nodes. Providing backoff boundaries requires a periodic event to be generated on the microcontroller on which 802.15.4 communication is initiated. Following the discussions in previous sections, this requirement leads to potential pitfalls in the context of portable implementation strategies.

Figure 7 depicts the operation of unslotted CSMA/CA. In the Figure, two examples, one of successful and the other of delayed packet transmission, are presented. Node 1 initiates a packet transmission sequence, and first assesses the RF channel during slot 6. The channel is free, and node 1 starts a contention period by delaying its transmission for a random number of backoff slots—5 in the example. This randomly calculated contention period allows nodes wishing to transmit at the same time to compete for channel access: the node with the shortest randomly-generated contention period will be the first to access the channel. At the end of its contention period, node 1 assesses the channel again. The channel is free and it commences the physical layer transmission of its packet. In contrast, node 3 initiates a packet transmission sequence during the contention period of node 1 (in time slot 8), and assesses the channel, finding it free. It initiates a 6-slot contention period, which ends at slot 13, at which point node 1's transmission is still ongoing, as indicated by the dashed line in the Figure. At the end of its contention period therefore, node 3 assesses the channel and finds it occupied. It therefore initiates a new contention access period, this time with a duration of 9 slots. At the expiration of this second contention period, the channel is free and the physical layer transmission of the packet is performed, starting in slot 24.

4.1. Typical 802.15.4 MAC implementations

In a typical 802.15.4 MAC implementation, when upper layers request services from the MAC layer, dedicated MAC tasks are added to a MAC scheduler queue. A dedicated MAC timer is

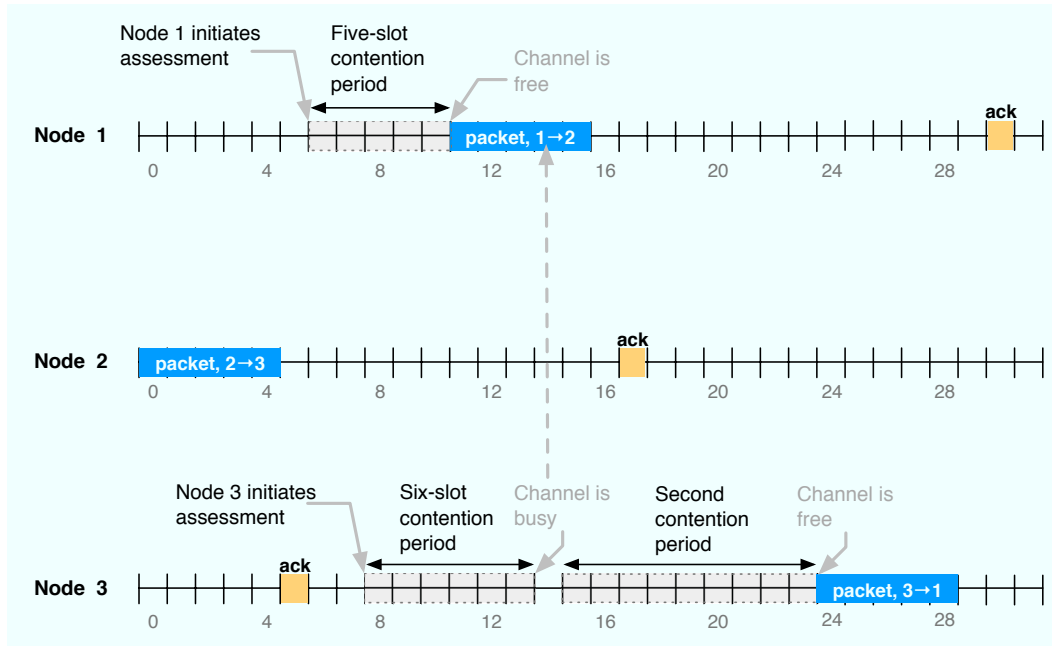


Figure 7. Transmission of frames using carrier sense multiple access with collision avoidance (CSMA/CA).

configured to generate *tick interrupts* at intervals of $320\ \mu\text{s}$. At each tick interrupt, the ready-to-run MAC task with the highest priority is run by the MAC *tick interrupt service routine (ISR)*. Microcontrollers executing such a MAC implementation thus need the ability to nest interrupts (i.e., accept interrupts during interrupt processing), since new high-priority events might occur while processing an extant MAC task. For example, MAC events such as the reception of a *start of frame delimiter (SFD)*, need to be processed immediately since they rely on short reaction times.

For example, the TI CC2420 transceiver uses two pins, **FIFO** and **FIFOP**, to indicate the state of its receive FIFOs. The **FIFO** pin is asserted when the first byte of a packet is placed in the receive FIFO; the **FIFOP** pin is similarly asserted when the last byte of a packet is received. A high latency in retrieving packets from the receive FIFOs might lead to FIFO overruns and thus, to lost packets. Without interrupt nesting, the microprocessor might be running a low priority interrupt service routine when the **FIFO** and **FIFOP** lines are asserted; it therefore becomes impossible for these higher-priority events to preempt the current ISR. Such issues affect portability, as implementations which are dependent on interrupt nesting must often be adapted when moving across platforms. Once an interrupt-based MAC implementation

is adopted, a later port of the system software to a task-oriented microkernel thus becomes difficult.

4.2. Overview of 802.15.4 MAC implementation challenges

Some of the challenges in implementing an IEEE 802.15.4 MAC may be generalized to other MAC implementations, since they relate to MAC state machine execution, and the control of both microprocessor and radio hardware. The challenges include:

- Integration of MAC state machine execution into a software execution environment.
- Reduction of a MAC implementation's influence on other software components.
- Usage of platform hardware abstraction libraries to control communication interfaces.
- Analysis of the effects of a microcontroller architecture (e.g., 8-, 16- and 24-bit microcontrollers) on data management.
- Striking a balance between timing requirements and available timer resources.
- Understanding and utilizing the MAC-related functions provided by a radio transceiver.
- Evaluating memory management services to build an optimized buffer manager.

4.3. Porting an IEEE 802.15.4 MAC from the Atmel ATmega128 to the NXP CoolFlux DSP

The NXP CoolFlux, a 24-bit DSP designed for mobile audio and embedded sensing applications, and the Atmel ATmega128, an 8-bit microcontroller, served as the target processors for the portability implementation study.

Both processors were employed in platforms containing a Chipcon CC2420 IEEE 802.15.4-compliant radio transceiver [37]. A reference MAC implementation from TI/Chipcon, targeted to a variant of the ATmega128 served as the starting point for the software portability investigation. The original reference implementation operated without OS support, and employed MAC tasks activated from within periodic timer interrupts. These MAC tasks were organized as state machines, and employed a *task information structure* data structure, placed in one of four FIFO queues with different priorities, to drive execution.

The Atmel and NXP low-power processors have differences in their hardware and software runtimes, summarized in Table III. Analyzing the constraints and optimization opportunities coming from these provided a good indication of what aspects of porting would need special attention. For example, from Table III, we observe that the less elaborate timer and interrupt capabilities of the CoolFlux will require care when porting to the CoolFlux, while the larger memory and possibility of running an operating system kernel (FreeRTOS) present opportunities for integrating the MAC software with little influence on other software components.

4.3.1. Interrupt-driven control of SPI interface

The software execution environment available on the CoolFlux DSP—the FreeRTOS microkernel—allows interrupt-driven control of hardware peripherals. The processor's

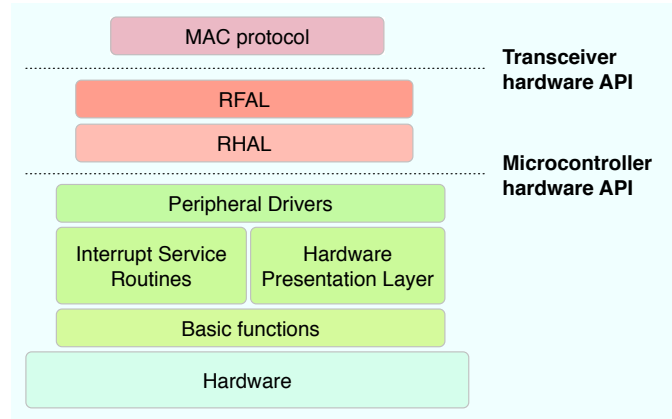


Figure 8. Radio driver for the CC2420 transceiver, comprising a low-level hardware dependent component, the *radio hardware abstraction layer (RHAL)*, and a high-level component, the *radio functionality abstraction layer (RFAL)*.

interfacing to the CC2420 radio via SPI was redesigned to use peripheral queues and interrupts to handle data transfer by hardware, in parallel to application processes, preventing hardware-dependent software polling of low-level SPI control bits indicating SPI communication status.

The gains of using the low-latency SPI interface were overshadowed by the latency introduced by such interrupt-driven SPI communication control. Latency is directly related to the time taken to save the context of the running FreeRTOS task, access the SPI queue to retrieve the next byte to send, and copy the data into the SPI data register for transmission. The direct effect is a throughput limitation in terms of packets per second that the node is able to transmit, as shown by Österlind et al. [27]. Latency in delivering data to the CC2420 is furthermore critical to avoid radio transceiver transmit FIFO underflow issues. A polling (rather than interrupt-driven) implementation on the other hand was observed to provide faster SPI transactions for applications requiring high throughput; portability via interrupt-driven control of the SPI therefore decreases the performance of the communication.

4.3.2. Creation of a transceiver driver

MAC software needs to combine low-level hardware access to control radio transceivers with the execution of higher-level functionalities. To separate the transceiver-hardware-dependent portions of the MAC from general protocol processing, a *radio driver* was developed (Figure 8). The radio driver comprises a low-level hardware-dependent component, the *radio hardware abstraction layer (RHAL)*, and a high-level component, the *radio functionality abstraction layer (RFAL)*, to serve as abstractions for the radio hardware and radio functionalities.

The mechanisms to send and receive RF packets require basic control over the SPI and *general purpose input/output (GPIO)* peripherals. OS peripheral drivers are used to prevent replication of code amongst processes and low-level access to peripherals' control and data registers, decoupling the MAC from the underlying microcontroller hardware.

The RHAL implements an API to configure the radio transceiver, pass data from the microcontroller to the radio transceiver, and initialize the interrupts used for handling packet reception. The RHAL was implemented on top of the provided OS peripheral drivers, serving as an abstraction for transceiver-specific hardware functions. The RHAL provides the RFAL with all the capabilities offered in hardware. An example of a macro offered by the RHAL is a function that allows the configuration of a transceiver internal register.

The RFAL, which is implemented on top of the RHAL, exposes transceiver functionality to the application or upper protocol layers. An example of such functionality is a function that an upper layer (e.g., the MAC) will call to set the channel on which communication will be scheduled. The way such a function is realized is generally dependent on both the way the transceiver is controlled (e.g., via transceiver register configuration) and the way microcontroller peripherals are controlled (e.g., via peripherals register configuration and control). By creating a radio driver, the upper layer is decoupled from different transceiver and microcontroller configurations.

4.3.3. Task-based MAC implementation

When several functions must be executed with different timing characteristics, dividing applications into components which are executed by independent tasks simplifies programming. The MAC implementation on the CoolFlux DSP provides the opportunity to have a microkernel task execute the MAC protocol with a task-status controlled by MAC events and timer interrupts. Such an implementation approach makes it portable to microcontrollers that do not allow nesting of interrupts, and reduces the influence of the MAC interrupt-driven implementation on other software components. The interrupt service routines can now be reserved for the most time-critical operations, such as the recording of timestamps or updating of state variables on the occurrence of various system events. These state updates are achieved in FreeRTOS through the use of semaphores, and they result in the tasks running above the kernel getting blocked or released to run.

4.3.4. The dual scheduling approach

The MAC task, like other tasks executing under the control of the operating system, is scheduled on the basis of the tick interrupts of the OS timer. Additionally, timers are used in the MAC for time-stamping and for coordinating channel access at backoff slot boundaries. Therefore, when the MAC is executed by an OS task, the OS needs to provide both timestamping and interrupt capabilities to schedule MAC functions. To cope with different timing and hardware support across platforms, the presented implementation of the MAC relies on a *dual scheduling* approach.

In this approach, one timer generates OS ticks for the microkernel scheduling while a second one generates ticks for the MAC scheduling, *but only when needed*[‡]. This design is motivated by

[‡]With more powerful timer engines, a single timer can generate the two ticks, e.g., using a single counter and two capture/compare registers, as described in Section 3.2.1.

the need for a microkernel tick interrupt independent of the MAC operation, so that the MAC can be ported to new software systems with minimal timing reconfiguration. The microkernel can therefore configure a tick interrupt at a chosen timer interrupt frequency, independent of (and, e.g., with much finer precision than,) the operating system's task-scheduling timer interrupts. Having an additional source of timer interrupts, at a granularity matched to the constraints of the MAC protocol (e.g., a multiple of the $16\mu\text{s}$ symbol duration), which can be deactivated when the MAC task does not require it, is beneficial to both ease of implementation and energy-efficiency.

4.3.5. Implementation

The IEEE 802.15.4 MAC was implemented within the FreeRTOS microkernel [4], following the dual scheduling architecture, for the SAND hardware platform [21]. The SAND node provides:

1. An NXP CoolFlux DSP [32] with 64 K x 32 bytes program memory and 64 K x 24 bytes data memory. It is equipped with two 16-bit timers, and runs at 20 MHz. SPI and GPIO interfaces are provided.
2. A port of the FreeRTOS operating system, with real-time preemptive multi-tasking [4].
3. A Chipcon CC2420 radio transceiver (2400–2483 MHz, 250 kb/s) [37].

FreeRTOS tasks are user-level tasks under control of the FreeRTOS microkernel. In the microkernel-based implementation of the MAC, the MAC is run by one FreeRTOS task, which we call the FreeRTOS MAC task. The FreeRTOS MAC task has the highest priority of all FreeRTOS tasks, in order to satisfy timing constraints related to network wireless communication. When other tasks with more strict timing constraints are needed, like time synchronization, the task priorities must be set accordingly. Two FreeRTOS binary semaphores are used to control the operations for packet transmission (TX) and packet reception (RX). First, the FreeRTOS MAC task checks the `FRTOS_MAC_TX_semaphore` semaphore for controlling transmit operations; when packet transmission is not required, the FreeRTOS MAC task cannot take the `FRTOS_MAC_TX_semaphore` semaphore. Next, the FreeRTOS MAC task checks the `FRTOS_MAC_RX_semaphore` semaphore in case a received packet is to be retrieved. In case no packets are received, control returns and the FreeRTOS MAC task loops back.

Applications that need to access the MAC can use a set of primitives defined by the 802.15.4 standard. As introduced in Section 2, transmission of data or command frames must happen 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, as shown in Figure 9. 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, and driven by information in a *task information structure*, as described in previous sections. 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 $320\mu\text{s}$. On interrupts, the MAC timer ISR releases the `FRTOS_MAC_TX_semaphore`, thus liberating the blocked FreeRTOS MAC task.

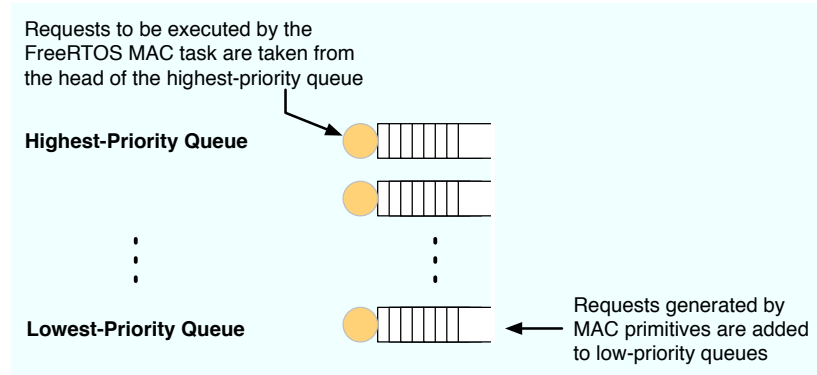


Figure 9. Execution of MAC functions as requested by MAC primitives.

On packet reception, the CC2420 radio 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_RX_semaphore` that blocks the FreeRTOS MAC task. The ISR terminates and the microkernel scheduler checks whether new microkernel 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.

Figure 10 summarizes the successive steps executed for processing packet transmission and reception. The different software triggers are highlighted to emphasize the software constructs and the relationship between the components of the software system.

4.3.6. Portability and performance evaluation

The challenges faced in transforming the baseline MAC implementation described in previous sections, into a portable implementation, included:

- Integration of the MAC state machine execution as an independent software component.
- Usage of hardware abstraction libraries to configure and control communication interfaces (e.g., SPI) and the transceiver, as well as software constructs independent of underlying hardware services, to prevent platform specific implementations.
- Separation of *MAC protocol* code from *transceiver driver* code.

The task-based MAC design presented in this article addressed the aforementioned challenges. The presented MAC implementation was subsequently ported, in approximately three months to a TelosB-like node (a TI MSP430 series microcontroller with a CC2420 RF transceiver). The MSP430 port in itself only involved compiler adaptations and timer settings. Due to the lower clock frequency of the MSP430 compared to the CoolFlux DSP, a large part of the porting effort involved optimizing the medium access mechanisms under heavy transmission load.

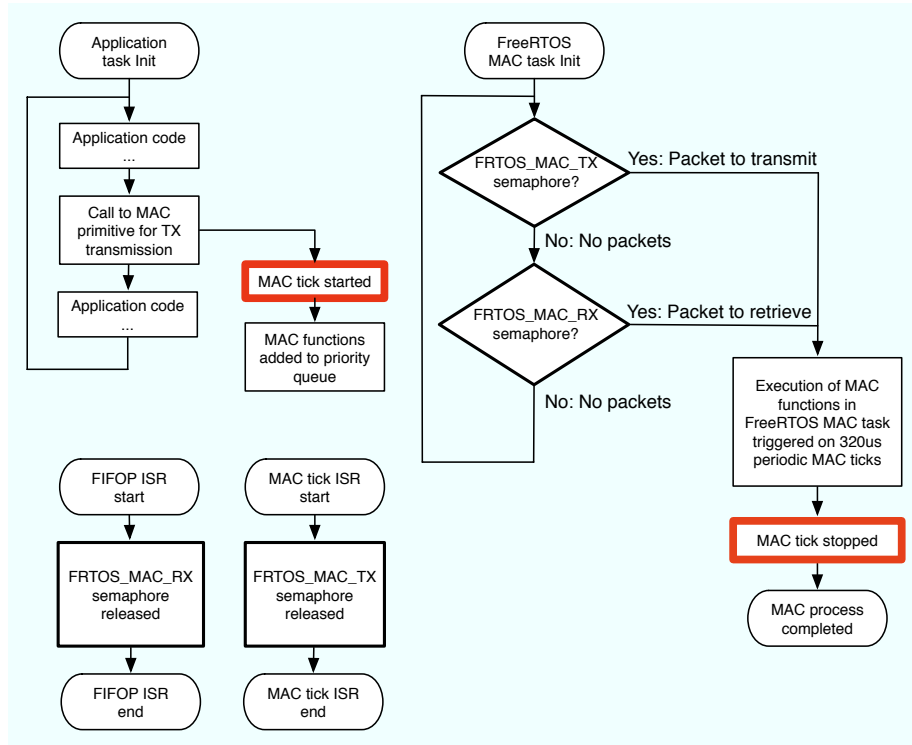


Figure 10. Triggers and relationships between software components for processing packet transmission and reception. FIFOP and MAC tick interrupts release MAC control semaphores, driving the execution of MAC code within the FreeRTOS MAC task.

The presented MAC implementation is currently used for a wide range of applications. The design permitted fast prototyping of new applications with different requirements, such as timing and process control. For example, a stroke rehabilitation application [42] required synchronized data sampling on different nodes to be able to meaningfully correlate data at the monitoring station. Time synchronization and distributed task synchronization [1, 2] have also been successfully implemented within the software environment comprising the task-based MAC implementation. As a further example, an emotion-sensing application [41] had requirements in terms of process control to enable on-node processing; the presented software environment permitted an easy integration of application software with no influence from the MAC process execution on task execution. Finally, a lightweight IP stack with a 6LoWPAN adaptation layer was seamlessly integrated within the presented software environment [33].

The current version of the MAC software includes support for both beacon and non-beacon networks. Two 128 B buffers, for outgoing and incoming packets, are used. The code size is approximately 11 KB and the RAM usage is approximately 2.3 KB. The original memory footprint of the reference MAC implementation was 21 KB for code and 2 KB RAM, with a typical additional requirement for three 200-byte RAM-based buffers. The code size reduction

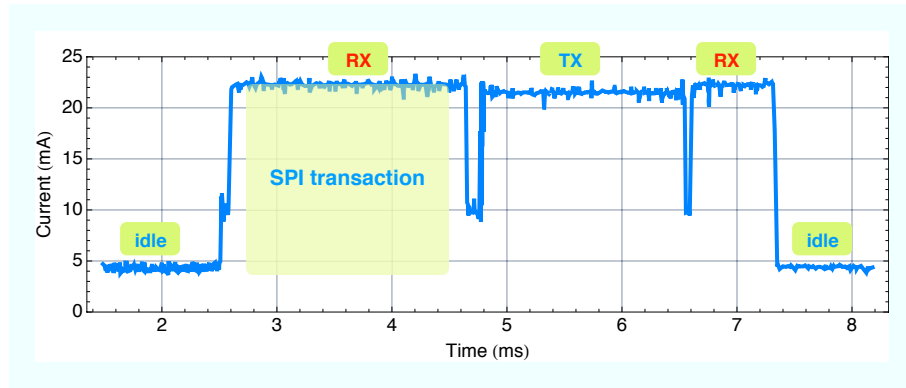


Figure 11. Energy and time duration involved in the different steps of a packet transmission realized with the presented task-based MAC design

for the CoolFlux DSP is related to the Target Compiler [35], a variant of which was specifically implemented for the CoolFlux architecture.

The best MAC payload throughput achieved on the SAND platform, using the portable implementation, was 86 kb/s. Each packet transmission included an 11 B MAC header, a 2 B MAC footer, and a 60 B MAC payload. Including the MAC header in the calculation, the throughput is 105 kb/s. This result is far below the theoretical upper bound of 220 kb/s given by Österlind et al. [27]. Our results are also lower than the throughput of 140 kb/s they achieved with a Tmote Sky node running the Contiki operating system under similar conditions. As another comparison, a data rate of 127 kb/s with Jennic JN5121 wireless modules was reported elsewhere [16]. To understand the origin of the lower throughput, we analyzed our software constructs to determine the cause of the poor throughput, and to ascertain whether the results were intrinsic to the dual scheduling design, or to its implementation.

Österlind et al. claim that the bottleneck in single-hop transfer on LR-WPAN platforms is neither the operating system nor clear-channel assessment, but rather, *serial peripheral interface (SPI)* packet copying [27]. SPI data transactions are required between the microprocessor and the transceiver to load the transmit FIFO with data, and can also be triggered in the receive path by interrupts, or by sampling the status of the interface (i.e., polling). Figure 11 shows a timing measurement realized on the SAND platform, with an interrupt-triggered SPI transaction. The SPI transaction happens during the first receive (RX) period, and lasts approximately 2 ms, representing one fourth of the total transmission duration. In an interrupt-triggered SPI transaction process, overhead comes from the context switch happening on each SPI interrupt and from the access to memory to fetch the next byte to transmit. Often, the overhead introduced by this method is longer than the transfer time. Consequently, for throughput purposes it is significantly faster to use polling.

An advantage of using an interrupt-driven approach is the decoupling of application code from communication code, as calls made to the communication hardware proceed

asynchronous to application flow. This results in an interleaving of communication with application execution, and fewer processor cycles are wasted in polling. For these reasons, we made both interrupt-triggered and polled SPI control available to application developers. A developer may thus choose to optimize for either latency/energy/throughput, or process multitasking according to the needs of their applications. In our implementation we chose to optimize processor efficiency at the expense of communication latency and throughput. If available on a given microcontroller, DMA facilities could be used to further reduce CPU overhead, thereby increasing the SPI throughput.

The FreeRTOS context switch latency on the CoolFlux DSP was measured to be $17\mu\text{s}$. During transmission using SPI polling, FreeRTOS context switches happen at each MAC tick interrupt, every $320\mu\text{s}$, and at each FreeRTOS tick interrupt, every 1 ms. With approximately four ticks per millisecond, a FreeRTOS context switch of $17\mu\text{s}$ represents an overhead of about 7% on the available processing power, whenever the network stack is active; when the MAC queues are empty, and no MAC ticks are required, this is reduced to less than 2%. Overall, our results show that the lower throughput performance mainly depends on the processor architecture and SPI interface mechanism.

5. Practical Guidelines for Portable MAC Implementations

From our experience porting MAC and transceiver interface software across hardware platforms, we observed that a small set of guidelines, if diligently followed, could significantly reduce the porting effort.

5.1. Software environment for MAC execution

Separation of software into independent threads of control is important for portability. Employing a single thread of control, e.g., placing the entire MAC state machine and application code within a single interrupt handler, combining the functionality of conceptually independent processes, makes software modification difficult. For example, a small code change could introduce changes in program timing which could in turn affect many functions unrelated to the code change. Furthermore, when employing a single thread of execution, all tasks embedded within this thread must be executed sequentially; priorities are then not easily given to subtasks in the sequential execution.

A single-thread execution environment therefore requires an interrupt-based MAC implementation to execute the MAC state machine, where MAC tasks are sequentially executing within interrupts; such implementation prevents porting to microprocessors not allowing nested interrupts and interrupt prioritization. Our experience has shown that integrating a MAC protocol implementation within an operating system tackles these issues, co-locating the MAC with other software components while avoiding code interdependency, as described in Section 4.3.

5.2. Management of interrupts

Implementing MAC-related routines inside interrupt service routines influences portability, since ISRs are low-level routines independent from MAC processes. Providing an interrupt subscription mechanism allows programmer-defined actions to be realized within interrupts. MAC routines, device drivers, and peripheral drivers can subscribe to a specific interrupt source, registering routines to be executed when the interrupt fires. Using such an interrupt subscription mechanism enables MAC code to be separated from actual interrupt handler code. Such approaches are commonplace in traditional operating systems.

5.3. Hardware abstraction

MAC implementations with direct hardware control might be able to exert more control over memory usage and latency, as illustrated, in Section 4.3.1, with the example of polling implementations for SPI interfaces. During ports to new platforms however, hardware-specific code is often difficult to find. It is thus beneficial to isolate all hardware-dependent code into separate modules, serving as the innards of a *hardware abstraction layer*. By using a unified hardware interface, provided to all system processes, MACs may be decoupled from the underlying microcontroller hardware. The use of a *radio driver* further extends such decoupling to radio transceiver hardware, separating a MAC protocol's transceiver-independent state machine from transceiver configuration and control.

5.4. Programming recommendations

Many texts on programming practice exist; notable examples of particular relevance to systems programmers include Kernighan and Pike's *The Practice of Programming* [17] and van der Linden's *Expert C Programming: Deep C Secrets* [40]. These texts provide a wealth of valuable information and insight. A few items, broached in the aforementioned texts and elsewhere, are worthwhile elaborating as they were the primary source of avoidable work in our porting efforts.

Compiler portability: strict ISO C90. An important item of caution, is the use of strict ANSI C, without taking advantage of specific extensions that compilers for a given platform may provide. For example, one component in our porting efforts was a time-synchronization algorithm whose implementation took advantage of hardware support on the original implementation platform, of fixed-point real-valued types, via compiler extensions. These extensions, which provided a new `fixed` data type, required a rewrite of the said algorithm.

Most compilers provide compile-time flags for checking whether their input strictly adheres to ISO C90, and the use of such flags is highly recommended during development of code intended to be ported across platforms. For example, the GNU Compiler Collection (GCC) provides the combination of compile-time flags `-ansi -pedantic`, which will cause non ISO C90 programs to be rejected at compile time.

No inline assembly language. In tandem with the above, it is almost always possible, and always advisable, to eschew the use of inline assembly language. Routines that need to

be implemented in assembler because they may require access to machine registers or specific instructions, should be placed in a assembly language source file, compiled separately, and linked with the C source objects at link time.

Central configuration header file. A significant impediment to many porting endeavors, is the spreading of configuration information, constants, data structure definitions, and global variables, across multiple source and header files; such scattering of information is seldom necessary. Many of these suggestions should be common knowledge to any competent programmer, and we only highlight practices that we believe deserve special reiteration.

Source `.c` files should contain source code that will be compiled into object `.o` files, and header `.h` files should contain data structure and constant definitions, and not procedure definitions. When the use of global data structures are necessary, they should be placed in the most central location of the system implementation, e.g., in the source file containing the system initialization routines, and `extern` references made in the source files where they are referenced. It is often furthermore desirable to group such items into a single structure, making the set of global state items self-evident.

Source code tree structure. The layout of a source tree can also have a significant effect on the time and mental energy required to port a software system. In this context, while structure in the tree layout is desirable, too many layers in a source tree are undesirable. The source tree should be divided, broadly, into (1) portable, platform-independent C code and header files; (2) processor architecture-specific code and headers; (3) platform-specific code and headers.

6. Summary and Concluding Remarks

The implementation of transceiver software and medium access control (MAC) protocols for resource-constrained embedded systems poses many challenges. These challenges range from the constraints on memory and compute resources typical of these platforms, to the diversity of microcontroller hardware architectures available in the domain, and the range of properties and behaviors exhibited by the plethora of radio frequency transceiver architectures. These challenges can however be surmounted with a careful approach to the design and implementation of transceiver interface software and MAC protocols.

This article provided an overview of the challenges facing transceiver software and MAC protocol implementers for LR-WPAN platforms, provided a study of different implementation architectures that might be pursued, and detailed our experience with transceiver software and MAC protocol implementations on a diverse set of platforms.

The three primary challenges to implementing medium access control protocols that are portable across radio transceivers and system platforms, are the diversity in radio transceiver interfaces and feature sets (the “bottom” interface of the MAC), the variety of system interfaces between the rest of the network protocol stack and the MAC (the “top” interface of the MAC), and the different interfaces to system resources such as memory allocation and synchronization primitives (the system software interface of the MAC); this article provided implementation recommendations built from our experience porting an IEEE 802.15.4 MAC across three platforms. One direction of ongoing research, is the definition of a standard for the “top”

and “bottom” interface that may be employed across future MAC protocols as well as across new transceiver architectures.

ACKNOWLEDGEMENTS

We are grateful to Peter van der Stok for his contribution to the design of the task-based MAC implementation. Albert Rietema, Peter van der Meer, Elmahi Adam and Tao Xu were instrumental in realizing the software implementation of the MAC. This work was supported by the EU IST program *WASP IST-034963* and by Science Foundation Ireland under grant 07/CE/I1147.

REFERENCES

1. M. Aoun, J. Catalano, and P. Stok. Distributed task synchronization in wireless sensor networks. In *EWSN '09: Proceedings of the 6th European Conference on Wireless Sensor Networks*, pages 150–165, Berlin, Heidelberg, 2009. Springer-Verlag.
2. M. Aoun, A. Schoofs, and P. van der Stok. Efficient time synchronization for wireless sensor networks in an industrial setting. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 419–420, New York, NY, USA, 2008. ACM.
3. Atmel Corporation. *AT86RF230 Low Power 2.4 GHz Radio Transceiver for ZigBee and IEEE 802.15.4 Applications*.
4. R. Barry. FreeRTOS™.
5. 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. *Mob. Netw. Appl.*, 10(4):563–579, 2005.
6. L. Burgy, L. Réveillère, J. L. Lawall, and G. Muller. A language-based approach for improving the robustness of network application protocol implementations. In *26th IEEE International Symposium on Reliable Distributed Systems*, pages 149–158, Beijing, China, Oct. 2007.
7. C. L. Conway and S. A. Edwards. Ndl: a domain-specific language for device drivers. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 30–36, New York, NY, USA, 2004. ACM.
8. A. Corporation. *ATmega128 8-bit Microcontroller with 128K Bytes In-System Programmable Flash*.
9. Crossbow. *TelosB Mote TPR2420*.
10. A. Dunkels, B. Gronvall, and T. Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.
11. A. El-Hoiydi, J.-D. Decotignie, C. Enz, and E. L. Roux. Poster abstract: wisemac, an ultra low power mac protocol for the wisenet wireless sensor network. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 302–303, New York, NY, USA, 2003. ACM.
12. Freescale Semiconductors. *802.15.4 MAC PHY Software Reference Manual Rev. 1.6*, 2008.
13. J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.
14. Institute of Electrical and Electronics Engineers. IEEE standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks specific requirements part 15.4: wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs). *IEEE Std 802.15.4-2003*, pages 1–670, 2003.
15. K. Jamieson, H. Balakrishnan, and Y. Tay. Sift: A MAC protocol for event-driven wireless sensor networks. In *EWSN '06*, pages 260–275, Zurich, Switzerland, Feb. 2006.
16. Jennic. *Calculating 802.15.4 Data Rates*, 2006.
17. B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999.
18. E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable tcp in the prolac protocol language. *SIGCOMM Comput. Commun. Rev.*, 29(4):3–13, 1999.
19. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

20. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. pages 115–148, 2005.
21. N. E. M. Ouwerkerk, W.F. Pasveer. Sand: a modular application development platform for miniature wireless sensors. In *Proceedings of the Body Sensor Networks Conference (BSN'06)*, 2006.
22. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.
23. F. M  rillon, L. R  veill  re, C. Consel, R. Marlet, and G. Muller. Devil: an idl for hardware programming. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.
24. R. Min and A. Chandrakasan. Mobicom poster: top five myths about the energy consumption of wireless communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(1):65–67, 2003.
25. Moteiv. *Tmote Sky Ultra low power IEEE 802.15.4 compliant wireless sensor module*.
26. E. I. Organick. *The multics system: an examination of its structure*. MIT Press, Cambridge, MA, USA, 1972.
27. F.   sterlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *Proceedings of the Fifth ACM Workshop on Embedded Networked Sensors (HotEmNets 2008)*, June 2008.
28. R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from bell labs. *Computing Systems*, 8(3):221–254, Summer 1995.
29. J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.
30. I. Rhee, A. Warri  r, M. Aia, and J. Min. Z-mac: a hybrid mac for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 90–101, New York, NY, USA, 2005. ACM.
31. D. M. Ritchie and K. Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.
32. H. Roeven, J. Coninx, and M. Ade. Coolflux dsp—the embedded ultra low power c-programmable dsp core. 2004.
33. A. Schoofs, C. Daymand, R. Sugar, U. Mueller, A. Lachenmann, S. M. Kamran, A. Gefflaut, L. Thiem, and M. Schuster. Ip-based testbed for herd monitoring. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'09)*, 2009.
34. A. Schoofs, P. van der Stok, and P. Stanley-Marbell. Portability versus efficiency tradeoffs in mac implementations for microsensor platforms. *Embedded Systems Letters, IEEE*, 1(1):24–27, may 2009.
35. N. Semiconductors. *CoolFlux DSP C Programmer's Manual PLR-15542 version 5.0*, 2005.
36. M. Sooriyabandara, T. Farnham, C. Efthymiou, M. Wellens, J. Riihij  rvi, P. M  h  nen, A. Gefflaut, J. A. Galache, D. Melpignano, and A. van Rooijen. Unified link layer API: A generic and open API to manage wireless media access. *Computer Communications*, 31(5):962–979, 2008.
37. Texas Instruments Inc. *CC2420 2.4 GHz IEEE 802.15.4/ZigBee-ready RF Transceiver*.
38. Texas Instruments Inc. *CC2430 A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee*.
39. Texas Instruments Inc. *MSP430 Mixed Signal Microcontroller*.
40. P. Van der Linden. *Expert C programming: deep C secrets*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1994.
41. J. Westerink, M. Ouwerkerk, T. Overbeek, W. Pasveer, and B. de Ruyter. Probing experience—from assessment of user emotions and behaviour to development of products. In *Philips Research Book Series, Vol. 8*, 2008.
42. R. D. Willmann, G. Lanfermann, P. Saini, A. Timmermans, J. te Vrugt, and S. Winter. Home stroke rehabilitation for the upper limbs. In *Proceedings of the 29th Annual International Conference of the IEEE EMBS Cite Internationale*, 2007.
43. P. Winterbottom, S. Dorward, D. Presotto, H. Trickey, R. Pike, and D. Ritchie. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.
44. W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Trans. Netw.*, 12(3):493–506, 2004.

Table I. Frame processing and microcontroller-unit (MCU) interface facilities provided by a selection of contemporary LR-WPAN transceivers.

Transceiver	MCU interface	En-crypt	RSSI readout	HW CRC	Auto-ACK	Low-power modes	CSMA/CA
Freescale MC13191/13201	SPI	None	No	Yes	No	3	No
Freescale MC13192/13202	SPI	None	No	Yes	No	3	No
Ember EM250/EM260	I2C, SPI	AES	Yes	Yes	Yes	2	No
	UART						
Atmel AT86RF230	SPI	None	Yes	Yes	Yes	1	Yes
TI CC2430/2431	SPI, UART,	AES	Yes	Yes	Yes	3	Yes
TI CC2420	SPI	AES	Yes	Yes	Yes	3	No
TI CC2520	SPI	AES	Yes	Yes	Yes	2	No
Nordic nRF24L01	SPI	None	No	Yes	Yes	2	No
Nordic nRF24LU1	USB, SPI, UART	AES	No	Yes	Yes	3	No

Table II. Hardware properties relevant to MAC protocol software implementations, for a collection of microcontrollers used in contemporary LR-WPAN platforms.

Microcontroller	Bus width (Bits)	On-chip SRAM (KB)	Timers, capture/compare	Nested interrupts	HW CRC unit	HW FXP-math
TI MSP430F2xxx	16	1–120	≥ 2 , ≥ 2	Yes	No	No
TI MSP430F5xxx	16	16	5, ≥ 5	Yes	Yes	No
Atmel ATmega	8	4	4, ≥ 4	Yes	No	No
Atmel AT91SAM7S (ARM7)	32	4–64	2, ≥ 2	Yes	No	No
NXP CoolFlux DSP	24	192	2, 0	No	No	Yes

Table III. Hardware and software support differences between the Atmel ATmega128 and the NXP CoolFlux DSP.

	Atmel ATmega128	NXP CoolFlux DSP
Hardware differences		
Timer engine	4 timers, 4 capture/compare registers	2 timers, No capture/compare registers
Word size	8-bit	24-bit
Interrupt nesting	Yes	No
Interrupt prioritization	Hardware	Software
Memory	128 KB program memory, 4 KB RAM	64 K words program memory, 64 K words RAM
Software support differences		
Execution environment	Single-threaded	Microkernel multi-threaded (FreeRTOS)
Hardware access	Direct access	Access via high-level hardware abstraction
Interrupt handling	Classical	Interrupt subscription mechanism
Memory management	MAC-dedicated buffer management	Software system buffer management