snapMac: a Generic MAC/PHY Architecture Enabling Flexible MAC Design[☆]

Pieter De Mil[*], Bart Jooris, Lieven Tytgat, Jeroen Hoebeke, Ingrid Moerman[**], Piet Demeester

Department of Information Technology, IBCN research group, Ghent University - iMinds, G. Crommenlaan 8 box 201, 9050 Gent, Belgium

Abstract

Timing is a key issue in many wireless, lower-layer (e.g., physical and data link layer) communication protocols. Maintaining time-critical behavior while increasing MAC protocol complexity is the challenge for many MAC implementations. To comply with stringent time constraints, current MAC implementations typically require such a tight integration to the radio driver that they become one monolithic block of code with MAC-specific logic hard coded at the lowest firmware level. Execution of time-critical functions in the firmware is a good strategy, but results in limited flexibility for MAC designers because the radio driver is dedicated for specific MAC protocol logic. We propose "snapMac": a generic MAC/PHY architecture with a clean separation between the MAC protocol logic at the user level and the execution at the radio firmware level. Our generic programming interface enables more flexibility, an easy way to compose new MAC designs, and getting feedback from the radio capabilities We demonstrate the feasibility and performance of this architecture by implementing it on a resource-constrained wireless sensor node. The experimental evaluation shows, for example, that we can simultaneously keep the flexibility of a software ACK and meet the ACK timing

[☆]Readers are reminded to treat all the data as being strictly CONFIDENTIAL during the review process. Before publishing this manuscript we first need to file our patent application.

[*]Corresponding author

[**]Principal corresponding author

Email address: `Pieter@DeMil.be` (Pieter De Mil)

URL: `http://ibcn.intec.ugent.be` (Pieter De Mil)

constraints as specified in the 802.15.4 standard. We also achieve 97% (i.e., 218 kbit/s) of the theoretical 802.15.4 throughput. This new implementation approach for MAC / PHY interactions has potential to be applied in other domains (e.g., WiFi, software defined radio, cognitive radio, etc.). Demonstrating the portability of snapMac is future work. "snapMac" enables the design and execution of new MAC protocols in a *snap*.

Keywords: radio hardware abstraction, MAC/PHY architecture, reconfigurable, MAC, flexible, time accurate radio driver

1. Introduction

1.1. Situation and Problem Statement

More and more devices are being connected to the Internet (Internet of Things (IoT) vision [1]), enabling a wide range of applications having varying requirements. Libelium [2] - a Wireless Sensor Network (WSN) platform provider - lists more than 50 IoT applications, grouped in different domains such as Smart Cities, Smart Metering, Security & Emergencies, Retail, Logistics, Home Automation, etc. Communication is mostly wireless, although the underlying PHY technology can vary (e.g., IEEE 802.15.4, Bluetooth, WiFi). The PHY hardware sets the upper bounds on the data rate (e.g., 250 kbit/s for the 2.4 GHz 802.15.4 PHY) but it are the radio driver and the Medium Access Control (MAC) layer who play an important role to meet the dispersed set of application requirements [3].

Because of these diverse applications requirements, lots of MAC protocols [4] were proposed over the last decades. Even in the IEEE 802.15.4-2006 [5] standard, multiple options are available: beacon-enabled or not, (un)slotted CSMA-CA or Guaranteed Time Slots. Also in the 802.15.4e [6] standard multiple MAC techniques (time slotted channel hopping, asynchronous multi-channel adaptation, low latency deterministic networks, etc.) and frame formats (multi-superframe, slotframe, etc.) are specified to support a wide range of industrial and commercial applications. As a result, a MAC designer needs to implement a complex algorithm (with stringent time constraints) that can execute radio actions in a time accurate manner.

A MAC designer basically wants the freedom to compose any MAC design, with time-accurate execution and without the need for changing the radio driver firmware. Conversely, a firmware engineer basically wants to implement a performant driver, with a generic interface and without the need for making it MAC protocol specific. With *snapMac* we target both MAC designers and firmware engineers. In [7], an important requirement of the future data link layer is given in the definition: "*In order for IoT systems to achieve full interoperability, as well as the support of heterogeneous technologies (...), this data link layer must allow for diversity.*". Diversity means being able to design completely new MAC protocols, but also being able to change the MAC protocol at run-time in order to achieve interoperability with other devices. Ideally, the driver firmware is MAC agnostic and a generic interface is presented to a MAC designer. This will enable both reusability and portability.

1.2. Contributions

We have been working on WSN solutions since 2005 and experienced that we had to design various MAC protocols. In 2012, we published our pluralisMAC [8] framework. This allows for flexible switching between MAC protocols (so-called maclets). These maclets use shared functions for controlling the radio. We have designed the first version of pluralisMAC on top of a traditional radio driver (TinyOS CC2420 driver). This meant that most of the MAC functions were tied to the driver, and time-accurate execution was not possible. In a national project, iMinds and RMoni have designed a new sensor platform (RM-090[1], using the CC2520 radio chip). There was no CC2520 driver available, so we took this opportunity to design it ourselves, keeping in mind the lessons we had learned. We addressed this challenging problem by designing snapMac: a generic MAC/PHY framework for flexible MAC design. Such a framework must allow to achieve the required time accuracy but with a clean separation between MAC and radio driver, hereby increasing flexibility, adaptability and portability at both levels. To proof this, we have implemented the proposed architecture on a resource-constrained sensor node and evaluated key metrics like ACK timing, throughput, round trip time and energy consumption. As such the key contributions of this article are the following:

- We present a novel generic MAC/PHY architecture, implemented on a resource-constrained device.
- We describe in detail how this architecture works.

[1]RM-090 info: http://www.rmoni.com/en/products/hardware/rm090

- We show that MAC design is more flexible, easier and more time-accurate by using our patent pending *chain of commands* technique.
- We experimentally validate the implementation of the snapMac architecture both in terms of functionality and performance regarding all essential individual mechanisms and a Low Power Listening MAC protocol.
- We describe the portability and reusability options of snapMac, which are not demonstrated in this paper.

1.3. Paper Structure

The remainder of the paper is organized as follows. Section 2 presents the related work. In Section 3, we describe the goals that enable MAC protocols to use a generic programming interface, portable across multiple heterogeneous platforms without sacrificing time accurate execution of the protocol logic. In Section 4 we propose our snapMAC concept. In Section 5 we show how to use snapMAC from a MAC layer point of view. We demonstrate the operation in an asynchronous multi-channel receiver-based communication scenario. Section 6 presents the performance evaluation of our snapMAC implementation. Section 7 discusses the portability of snapMac (because porting to other platforms is work in progress). Finally, Section 8 states our conclusions.

2. Related Work

In this section, we highlight the bottlenecks on resource-constrained platforms, the related work in the WSN domain, and the related work in non-WSN domains.

2.1. Bottlenecks on Resource-constrained Platforms

We have identified the bottlenecks on a resource-constrained platform. A WSN platform is relatively constrained in term of resources (e.g., a class 1 node has 10 kB data size and 100 kB code size [9]). The processing power should be kept to the minimum to support the application and to guarantee the maximum node lifetime. The design of a generic MAC/PHY architecture should therefore cope with the resource scarcity, while still allowing sufficient flexibility.

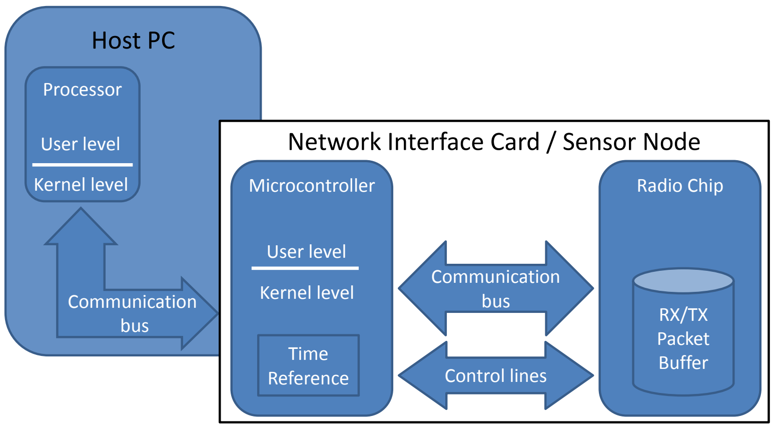


Figure 1: Typical setup of a Network Interface Card (NIC) or sensor node plugged into a host PC communication bus (e.g., PCI, USB, etc.). The NIC/sensor node typically has a microcontroller and radio chip interconnected via an on-board communication bus (e.g., SPI) and control lines. Unlike a NIC, a sensor node can operate without host PC.

The radio chip of a typical sensor node (or Network Interface Card), depicted in Figure 1, is responsible for transmitting or receiving data that is generated or consumed on the processing unit (processor on host PC, microcontroller on a stand-alone sensor node). This data needs to be transferred to the radio unit via a communication bus (e.g., SPI). When multiple peripherals are connected to a *shared* communication bus, unpredictable access timings and/or packet transfer rate are often unavoidable. On the TelosB (TMote Sky) platform [10], the CC2420 radio chip and the flash storage unit share the SPI bus with the processing unit. An SPI bus arbiter is needed to grant access to a specific slave unit. Hence, it might take some time to get access to the bus when a transfer to/from the flash storage device is active. The same is valid for the Zolertia Z1 platform [11], while on the waspmote platform [12] the communication bus is shared with the USB port. As a consequence it is difficult to tightly control the timing of radio actions on such hardware platforms (although it would be possible if the drivers for *all* the peripherals are managed by the same engine). We conclude that the communication bus between the radio and the processing unit introduces variable, unpredictable delays and jitter. We advise to use a dedicated communication bus to improve the time accuracy.

Another bottleneck is the fact that processes are executed sequentially on the microcontroller. It is unpredictable when exactly (i.e., microsecond precision) processes are executed in a non real-time operating system. User processes have a lower priority than kernel processes, but even kernel timers in a Linux-based OS cannot guarantee to execute processes precisely on the requested time, as the author in [13] states that "*a kernel timer is far from perfect, as it suffers from jitter and other artifacts induced by hardware interrupts, as well as other timers and other asynchronous tasks*". We conclude that too many interrupt sources or too long interrupt routines and asynchronous tasks (code reachable from an interrupt handler) may affect the time accuracy.

2.2. WSN domain

In the WSN domain, to the best knowledge of the authors of this paper, we have not encountered a solution that offers both the desired flexibility *and* time-accurate performance.

TinyOS [14] and Contiki [15] are two popular operating systems for WSNs [16], that do not provide real-time guarantees. In most OSes, there are basically two layers of execution (although there is no magic user/kernel boundary): user level (e.g., the code that is executed by **tasks** in TinyOS) and kernel level (i.e., the code that is executed by the Interrupt Service Routines (ISRs)). An ISR blocks all other tasks from running and should therefore contain only very short living functions. When an ISR is finished, the task scheduler selects which task is executed next. Typically it is advised to limit ISRs to only toggle a flag and copy the needed register information. Upon the end of an ISR a *system* task should finish the task. For a TinyOS implementation, one could use the concept of tasklets[2]. Our proposed framework is meant to be applicable on various Operating Systems. We have implemented snapMac without using such a tasklet mechanism, and demonstrated the high performance of our snapMac stack. In a future revision of snapMac, we might use the tasklet mechanism.

In Figure 2 we can see that the hardware management is typically done at the lowest level, and consists of three different functional blocks: the radio control, the bus manager and the interrupt manager. These Hardware Abstractation Layers (HAL) expose a Hardware Abstraction Interface that simplifies the hardware management. Typical functions exposed are enabling/disabling the hardware clock, request or release the bus, adjust some

[2]https://github.com/tinyos/tinyos-main/wiki/Tasklet-design-notes

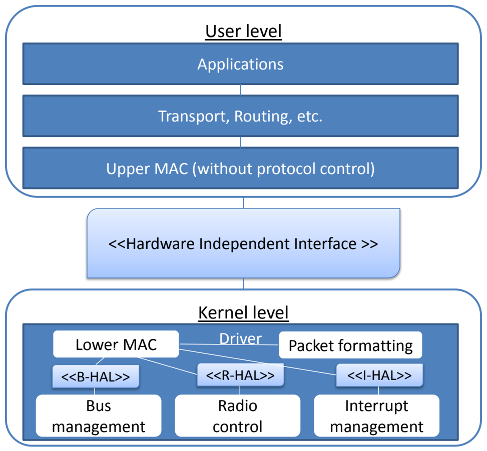


Figure 2: Traditional software stack divided between user level and kernel level, the latter contains usually a driver with a dedicated Lower MAC (protocol control) and fixed MAC frame formatting.

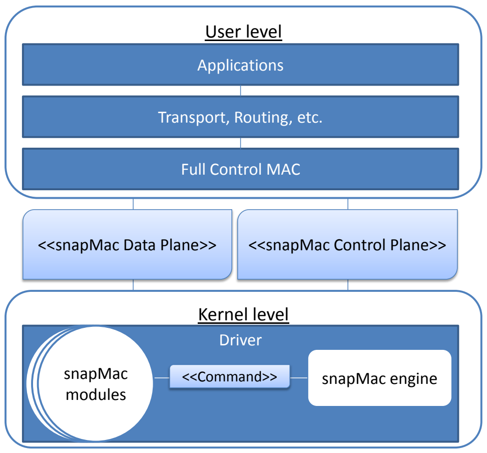


Figure 3: snapMac exposes two interfaces (data plane and control plane) to the Full Control MAC in user level. The snapMac engine and snapMac modules in the kernel level are MAC agnostic.

radio settings like channel, transmit power, etc. The MAC functionality builds on top of these interfaces and exposes a Hardware Independent Interface towards user space. This layer is therefore also called the Hardware Independent Layer (HIL).

In TinyOS, the basic HIL that must be provided for every platform is the *ActiveMessageC* configuration. This defines the message buffer format to be used, and has *send* and *receive* function calls. A *send* call is executed as soon as possible after the moment it is called. This results in an unpredictable time between the call, and the actual sending of the frame. In addition, the time accuracy of the call itself has a large spread. We have measured the delay between the requested transmit time, and the effective transmit time to be almost uniformly distributed between $10 - 20$ ms on a TMote Sky node running TinyOS. Hence a MAC designer who wants accurate frame transmission timings (in microseconds range) has to implement this in kernel space, hereby hampering reusability of the MAC code. Ideally, the basic (radio) functions are made available to the MAC designer, so that she has the flexibility to compose new MAC designs without changing the driver.

In standard TinyOS, the actual frame formatting is done inside the driver. We will show that we can give control on frame formatting to the MAC designer, hereby enabling both flexibility for the MAC designers (no need to adjust the driver) and portability for driver developers (no need to take into account specific frame formatting).

In [17], Hauer presents a platform independent IEEE 802.15.4-2006 MAC implementation (TKN 15.4) for TinyOS. It is stated that in most typical sensor node platforms, "these (ed., timing) requirements can practically not be met by a platform independent MAC protocol, rather they should be pushed from the MAC to the PHY, ideally to hardware". A common approach is to push time-critical operations from the MAC to the lower level radio control. In [17], the radio driver includes a part of a specific MAC protocol and exposes those MAC specific services via non-generic interfaces. We will show that it is possible to create a MAC protocol agnostic driver.

In [18], Steiner et al. analyzed traditional MAC protocols in order to obtain a *generalized* C-MAC state machine (some states are optional or can have different implementations) for three major categories: channel polling, scheduled contention, and time division multiple access. This C-MAC concept is targeted to these three MAC protocol categories and does not support fine-grained, time-accurate control on the radio functions (the Round Trip Time between two nodes is 79 ms). C-MAC makes it possible for easy configuring a MAC protocol that fits in one of the three categories, but it does not foster the creation of completely new MAC protocols.

In [19], a component-based MAC layer architecture (MLA) for implementing different MAC protocols is presented. MLA uses two types of components: "*High-level, hardware-independent components (channel poller, LPL listener, preamble sender, time synchronization, slot handlers, low level dispatcher) are aimed at supporting flexibility by allowing different MAC protocol features to be composed together in a platform independent manner. Low-level, hardware dependent components (radio power control, channel monitor, cca control, low-cost packet resending, low-latency I/O, alarms) provide abstract, platform independent interfaces to features otherwise specific to a particular radio or microprocessor platform. Though the implementation of these hardware-dependent components is inherently platform specific, they export interfaces which support the development of fully platform independent high-level components.*". snapMac does not provide hardware-independent components or low-level hardware-dependent components. Instead, it provides the basic commands which enable to create any desired functionality. For each of the hardware-dependent and -independent components (MLA) it is possible to create a snapMac command chain. MLA has used a different approach to solve the issues. In MLA, it is not possible for a MAC designer to know how long it takes to turn on the radio and to plan the execution time of a command (for example, transmit). For the TDMA MAC protocols, an async interface is exposed to the MAC designer. This will make the MAC control faster but not precise. No async interface is exposed to the user level in snapMac, but we do allow precise control of the (low-level) actions, in user level.

From the above analysis we can state that system developers try to achieve the required time accuracy leading to the typical kernel level / user level separation shown in Figure 2. All time-critical functions (e.g., sending an ACK) are implemented in kernel space, which basically pushes (traditionally) the full MAC (or at least the lower-MAC) implementation into the radio driver as it requires strict timing. In addition, a strong coupling between the radio driver and the in-kernel MAC is typically needed in these conventional approaches in order to achieve sufficiently low response times. The need for low response times - e.g. when an ACK message needs to be transmitted as a response to a successfully received frame - also results in the radio driver being frame type dependent. This implies that the reusability of both MAC implementation and radio driver is very low, as they are dependent on the implemented standard (or proprietary protocol) and also dependent on each other. Due to the large amount of work needed to build a MAC from scratch for every wireless standard and every new generation of radio hardware, the complexity and adaptability of current MAC protocols following such an approach is severely limited.

2.3. Non-WSN domains

Most work on flexible and/or time-accurate architectures has been done in non-WSN domains so far: Software Defined Radios (SDRs), Cognitive Radio, WLAN and overlay MACs.

SDRs (e.g., USRP platform [3]), offer spectrum agility, because the radio parameters like frequency band and modulation type can be reconfigured. SDRs implement most of the physical layer and link layer in software. Because of the intensive signal processing (mostly by using multiple processing units with interconnecting buses [20]), large delays (up to hundreds of microseconds [20]) and jitter are introduced. In [21], the measured re-

[3]http://www.ettus.com/

ceive latency ranges from 1 ms up to 30 ms. Obviously, these high latencies limit the response time and precise timing control needed in a MAC design (e.g., the default ACK timeout is 48 μs in 802.11b and 864 μs in the 2.4GHz 802.15.4). In [20], Nychis et al. present a split-functionality approach where a minimum set of core MAC functions are implemented close to the radio (e.g., FPGA processing on the hardware), while maintaining control on the host processing unit through a programming interface. They have concluded that time-critical radio or MAC functions should not be placed on the host CPU and that in order to be widely applicable, the control of the flexible MAC implementation should reside on the host.

A Cognitive Radio is based on SDR technology, and adds knowledge so that dynamic spectrum access becomes a reality. In this domain, aspects like adaptability and flexibility are very important. We agree with Ansari et al. [22] [23] that a MAC protocol implemented in a monolithic fashion with tight coupling to the underlying hardware restricts these aspects. The authors propose a decomposable MAC framework by defining a set of MAC functionalities (blocks) as a library. This way a wide range of protocols can be realized by combining these MAC blocks through a wiring engine, but designing completely new protocols is not supported.

In the WLAN (Wi-Fi) domain, Tinnirello et al. [24] have introduced a programmable wireless platform "Wireless MAC Processors"(WMP) that supports a MAC defined in terms of a Finite State Machine (FSM). This FSM consists of transitions between states. These transitions can be triggered by events (e.g., frame received). The transition will be executed if a certain boolean condition is TRUE (e.g., ACK on). Before completing the transition to the new state, an action (e.g., transmit ACK) can be performed. We like the idea of having an "instruction set" (actions, events, conditions) to compose an FSM that is executed by a MAC protocol agnostic engine. Unfortunately it is impossible to adapt a MAC protocol (FSM) at run-time, locally on the NIC. In WMP, an adapted MAC protocol (new FSM) needs to be recompiled on a remote machine and re-injected on the interface card. We will show that snapMac offers run-time reconfigurability of any MAC design without needing a remote machine.

In [25], Djukic et al. present a software TDMA MAC protocol, implemented in Linux user level, running over commodity 802.11 hardware. The focus is on tight synchronization of pairs of nodes. The 1.5 GHz laptop runs Linux with real-time extensions, which is necessary for precise software timers. The user space uses a real-time thread implementation. The authors want to test new standards (802.11s, 802.16 mesh protocol, etc.) and have chosen for an overlay approach because “it is unlikely that the equipment implementing these standards can be modified to develop new TDMA protocols”. In [26], Rao et al. present an overlay MAC layer (OML) solution. They want to test new protocols without changing the underlying MAC. Replacing this existing MAC layer is much harder because it is implemented “partly in hardware, partly in firmware, and partly in the device driver of the Network Interface Card”. This tight integration of a MAC with the driver is one of the reasons why we have made snapMAC MAC-independent. We can conclude that any overlay MAC is always limited by the interface provided by the underlying MAC layer (and consequently the PHY driver). A MAC protocol using snapMAC also runs in user level, but not on top a MAC specific driver. This way, it is much easier to implement any MAC.

The goals in these non-WSN domains are the same as ours (i.e., flexibility and adaptability), but this often goes together with a reduced performance. We have focussed on flexibility, adaptability, and performance. Furthermore, our proposed architecture aims to be universally applicable (portability) thanks to the clean separation between MAC logic and radio driver.

3. Goals of snapMAC

The move towards a generic, MAC protocol agnostic radio programming interface fostering the design of any MAC is the major goal of snapMAC. This radio programming interface (and architecture) should therefore present a solution to the constraints described in the introduction. In this section we propose our 5 goals, related to the problems of traditional MAC design.

We have shown that implementing time-critical operations in the MAC becomes increasingly problematic in user level. Pushing time-critical operations from the user level MAC to a lower level reduces the implementation complexity at user level. In traditional MAC design, a part of the MAC is running in kernel space, which makes debugging harder, and the risk of creating kernel lockups higher. Hence we would like to **design the MAC logic in user space**, with support to control the time-critical operations. This way the development and debugging are less complicated. Moreover this also increases the portability as different platforms / operating systems only have to supply the generic interface and execution environment.

A generic radio interface (API) must allow MAC designers to reuse the same interface across different hardware/software platforms and different radio chips (PHYs). This goal will enable **reusability** of MAC implementations on different platforms. Of course, radio chips for different PHYs may have divergent features. Being able to add these new features without changing the interface (API) is a big advantage. A traditional radio interface needs to be extended with new interface functions any time new features are available. This is not needed with a *generic* radio interface because the specific new command (e.g., for a new feature) is a parameter of that function call. This way, new commands (features) can be implemented in the driver, and made available to the MAC designer without changing the API.

We have seen dedicated radio drivers that are quite accurate only because they are designed and implemented for a specific MAC protocol (traditional concept). In our concept the radio driver must be MAC protocol and MAC frame format agnostic. This will advance **portability** of both the radio driver and the MAC protocol logic to other platforms.

Some actions require time-critical decisions which cannot be made at the user space MAC protocol level. For example, checking if the ACK request-bit is set in a header, copying the sequence number to an ACK frame and sending that frame within the same deadline as a hardware autoACK. Another example is the random back-off interval in the CSMA/CA algorithm. The user level MAC designer can set the constraints (e.g., minimum and maximum backoff exponent), but the random value will be determined in the hardware abstraction architecture. Any time-critical decision that must be made within 1000 μs should be done in kernel level. The architecture must support a generic mechanism to allow these **time-critical interactions**, without requiring MAC-specific radio driver extensions (as is done in traditional MAC design). Furthermore, it must be possible to implement a standard compliant MAC.

We also want to provide a flexible research platform for MAC designers. It must be possible to make per frame settings (channel, transmit power, use CSMA/CA or not, etc.) and time-accurate, run-time **reconfiguration** of any MAC design. With traditional MAC design, this level of flexibility is impossible. Our framework will speed up the prototyping/development of enhanced MAC protocols that may exploit new radio chip features. This will also yield a fair comparison of MAC protocols because the underlying radio driver is MAC protocol agnostic.

4. The snapMAC Architecture

4.1. Overview

Based on the goals proposed in the previous section we have designed snapMac, a generic MAC/PHY framework for flexible MAC design. A conventional upper MAC, running in user level, cannot reach sufficiently accurate timings (i.e., microseconds precision) and therefore a kernel level lower MAC with pre-defined protocol logic is usually needed. We think it would be better if the user level MAC can decide, in advance, when exactly an action/command (e.g., turn radio on, set channel, listen, check a bit in a frame, etc.) must be executed, without being constrained by the kernel level driver.

The user level MAC on top of snapMac, shown in Figure 4, uses two interfaces: *snapMacDataPlane* and *snapMacControlPlane*. The *snapMac-DataPlane* interface, exposed by the kernel level Data Plane Toolbox, is used for sending and receiving frames. The *snapMacControlPlane* interface, exposed by the kernel level snapEngine, is used for posting the user-defined MAC protocol logic to snapMac.

Through this Control Plane interface, one or more time-annoted chains (e.g., chain T) can be posted. A chain is composed of commands and is used to describe MAC protocol logic. Most of the commands are implemented in snapMac modules. A module can be a general toolbox (e.g., data plane and arithmetic) or a hardware abstraction for a specific chip (e.g., antenna switcher A HAL, radio B HAL). Each module exposes the *Command* interface to the snapMac Engine. Based on the timestamps associated with each posted chain, the scheduler will sort the chains. Via the dispatcher, each command is executed just in time by the suitable module.

4.2. Modules, States, and Commands

First, we will explain modules, states, and commands. In order to clarify some concepts, we have added examples that might be related to the PHY technology (e.g., 802.15.4) used in the radio chip.

A **module** is a software entity that contains the implementation of a command set. A module may also keep a state and each module has a library of commands. We currently have four modules in snapMac: Data Plane Toolbox, Arithmetic Toolbox, Radio B (e.g., CC2520) HAL, Antenna Switcher A HAL.

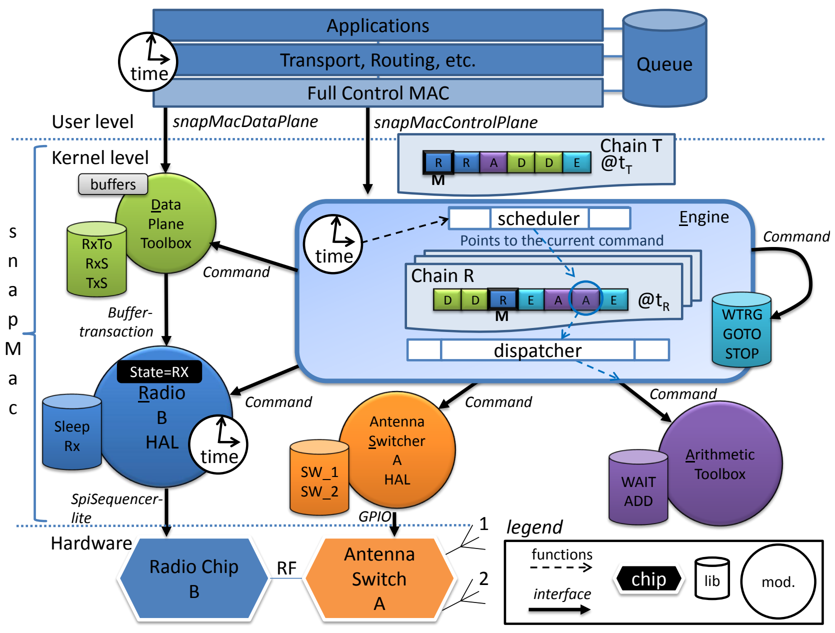


Figure 4: Our snapMAC architecture. The Full Control MAC (in user level) uses two generic interfaces: *snapMacDataPlane* for the data flow and *snapMacControlPlane* for the MAC protocol logic. Our "*chain*" technique is used to describe the MAC protocol logic. *snapMac* (one *engine* and n *modules*) is responsible for time-accurate execution of the user-defined protocol logic.

A **command** is a primitive action with an optional parameter and a certain command type. A command *may* change the state of a module after its execution. Example commands are: "turn the radio off" (module state changed) or "set auto ack" (module state not changed).

A **chain** is used for describing the MAC protocol logic as a sequence of commands. Based on the command types, it is allowed to jump to another command in the chain.

The module's **state** can be used to verify whether a command is allowed to be executed or not. For example, the command "turn radio on from off" can only be executed if the state before executing this command is "OFF". After successful execution of this command, the state will be "ON".

We further distinguish between stable states (e.g., OFF, SLEEP, ON, RX) and transient states (e.g., rxtx turnaround, tx, txrx turnaround). Commands must always start from a stable state and end with a stable state (if the module keeps state). The transient states only last for a short time. We have used this for the implementation of the “Transmit frame” (Tx) command. After a transmission the radio (e.g., Radio B = CC2520) will return to the stable state RX automatically, that is why we do not have a stable state TX in the Radio B HAL module. Before we can transmit a frame, we must check if the radio is in stable state ON or RX. Next, three more steps are required: switch the radio to tx (rx-tx turnarond), transmit the frame (tx), and switch back to rx (tx-rx turnaround). We will use subcommands for these steps. Such a sequence of one command followed by n subcommands is a *subchain*. The user level MAC designer can not call individual subcommands. It will be done automatically. A subcommand may start and end with a transient state, as long as the start and the end of the subchain have stable states. This is shown in Figure 5.

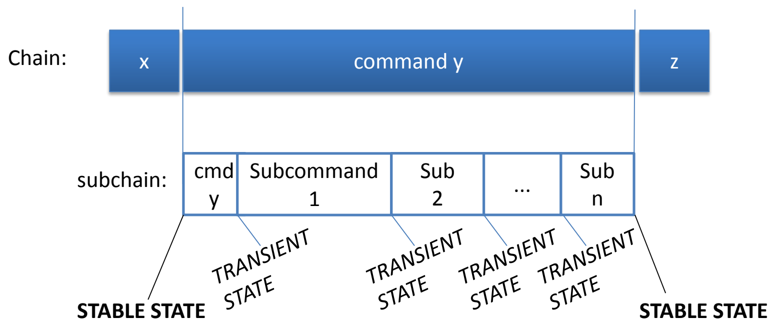


Figure 5: A command always starts from a stable state and ends in a stable state. A command y may have subcommands. In this case, the subchain starts with the command y, followed by n subcommands. A transient state is a temporary state, mostly until the execution of a short hardware-related action (e.g., rx-tx turnaround) has finished.

Each module will expose the narrow waist *Command* interface (see Listing 1) to the *snapEngine*. The *estimateTime2FinalizeCommand* function returns how long it will take to finalize (execution and optional transition) a specific command taken into account the parameter(s) of the command, based on a configurable lookup table that contains the timings. The *executeCommand* function will actually execute the command that is requested by the *snapEngine*. A module can also throw a *commandEvent* to the *snapEngine* if an event occurs. The *getCurrentState* function will return the current state of the snapMac module (if applicable).

```
command TimeStampT estimateTime2FinalizeCommand(CommandT
    commandId, ParamT param, StateT blockingState);

async command uint32_t executeCommand(CommandT commandId,
    ParamT param, StateT blockingState);

async event void commandEvent(EventT cmdEvent, CommandT
    commandId, ErrorT error, uint16_t info, TimeStampT ts);

async command StateT getCurrentState();
```

Listing 1: The Command interface exposed by each module (The keywords command, async and event at the beginning of the functions are nesC related keywords explained in [16], not to be confused with our snapMac Commands and Events)

Every module will have its own library of commands, but the narrow waist *Command* interface is reused for every module. Next, we will give more details on our two classes of modules: hardware abstraction modules and toolbox modules.

4.2.1. Hardware Abstraction Modules

The hardware abstratction modules contain commands that are hardware *dependent*. The RM-090 platform has a dedicated SPI bus between the processing unit (MSP430f5437) and the radio chip (IEEE 802.15.4 compliant CC2520). It also has an antenna switch chip connected via General Purpose Input Output (GPIO) lines.

In order to control these hardware chips, we needed to implement suitable hardware abstraction modules (drivers). This was done with separate modules for each chip. For example, the hardware abstraction for radio chip *B* is implemented in the Radio B HAL module. This module contains the implementation of radio related commands and tracks the state of the radio in order to verify whether a command can be executed or not. Similarly, the driver for the antenna switch chip *A* is implemented in the Antenna Switch A HAL module. These commands expose the features of the hardware.

The hardware abstraction modules must be able to communicate with the chip itself. If there is an SPI communication bus, this can be done via the *SpiSequencerLite* interface. We developed a new SPI driver, but this is out of scope of this paper. Controlling a chip via GPIO control lines is also possible via the *GPIO* interface. We did this for the antenna switch chip. Of course, the interface a firmware engineer will have to use depends on the design of the platform and new interfaces can be added easily.

The capabilities (maximum frame length, number of supported channels, configurable transmit power, etc.) of the underlying radio hardware are available in the header file of each Radio HAL module. It is the responsibility of the MAC designer to check the capabilities of the radio chip when an existing chain is reused.

4.2.2. Toolbox Modules

The toolbox modules contain commands that are hardware *independent*. This property makes them reusable across different platforms. Currently, we have two toolboxes: the Arithmetic Toolbox and the Data Plane Toolbox.

The Arithmetic Toolbox implements commands related to arithmetic operations like comparing variables, adding variables together, setting/checking a deadline, getting a random value, etc.

The Data Plane Toolbox buffers incoming and outgoing frames, hereby avoiding blocking of the radio HAL when multiple frames are received before the user level MAC is able to retrieve the frames and free the memory space. Additionally it can store multiple frames waiting to be transmitted. This toolbox exposes the *snapMacDataPlane* interface (Listing 2) directly to the user level MAC.

```
command error_t postTxFrame(ChainT chainId, uint8_t* frame,
    uint16_t size);

event void txLoadFrameDone(ChainT chainId, uint8_t* frame);

command error_t postDataBuffer(ChainT chainId, uint8_t*
    buffer, uint16_t size);

event void completedDataBuffer(ChainT chainId, uint8_t*
    buffer, uint16_t size, TimeStampT start);

event void droppedDataBuffer(ChainT chainId, uint16_t size,
    TimeStampT start);
```

Listing 2: The snapMacDataPlane interface exposed by the Data Plane Toolbox module (The keywords command and event at the beginning of the functions are nesC related keywords explained in [16], not to be confused with our snapMac Commands and Events)

The *postTxFrame* function will enable the user level MAC to post multiple frames (depends on the configurable buffer capacity) that need to be transmitted. Posting a Tx frame in the Date Plane Toolbox buffers can be done at any time. It is the MAC logic that will decide when a frame can be transmitted. At that time, a command implemented in the radio HAL module will get the next frame from the Data Plane Toolbox. The *buffer-Transaction* interface, exposed by the Radio B HAL module (see Figure 4), will be used for exchanging frames between the Data Plane Toolbox and the Radio HAL module. The *txLoadFrameDone* event will be signalled to the user level MAC when the *TxSignal* (TxS) command is executed in the Data Plane toolbox.

Similarly, we have functions and events for the receiving part. The *postDataBuffer* function will enable the user level MAC to give the Data Plane Toolbox a buffer where it can store the received data. The *completed-DataBuffer* event will be signalled to the user level MAC when the *RxSignal* (RxS) command is executed. If there is no buffer capacity left in the Data Plane Toolbox, the *droppedDataBuffer* event will be signalled to the user level MAC for every frame retrieved by the radio HAL module.

The *snapMacDataPlane* interface makes it possible for a user level MAC designer to use a user level message queue and post pointers to queue entries to the Data Plane Toolbox. This mechanism makes it possible to quickly integrate the snapMac solution with existing wireless stacks.

4.3. Accurate time reference

Our system must work on so-called “radio time”. This is the time used by the radio driver in order to meet the timings of the standard specification it wants to be compliant with. A stable clock could be derived from the radio chip itself (via the GPIO pins of the radio and a external clock input on the microcontroller) but the radio can only deliver a clock when it is active. This would take too much energy and is unfeasible in low-power applications. The symbol time (T*symbol*) (e.g., 16 µs) used in the radio chip must be a multiple of the microcontroller’s ”system time” (e.g., 1 µs). If we can match the system time with the radio time, we have a very big advantage for our time accurate driver. Other sensor boards often use a 32768 Hz crystal which can only achieve a T*sympbol* of 15.259 µs [27]. This is not accurate (4.6% error). For this reason we have not used the standard 32768 Hz crystal for our microcontroller. A 32000 Hz crystal was chosen, after confirmation from Texas Instruments that this was OK for the microconroller. By using a 32000 Hz ± 40 ppm crystal, we can map [4] this crystal on 1 µs sub-symbol

[4]This is how we have mapped our 32000 Hz crystal : 32000 Hz times 500 is 16 MHz; 16 MHz divided by 16 is 1 MHz, 1 tick every microsecond

ticks which is 16 times smaller than the 2.4 GHz 802.15.4 symbol time. This will allow us to define and execute the MAC protocol logic very accurately.

4.4. Protocol Logic through Chains

The user level MAC is not allowed to call the modules' commands directly. Instead, the MAC protocol logic is defined by the MAC designer by composing commands into so-called chains. The purpose of a chain is to have time accurate execution of the MAC protocol logic. Suppose we want the transmission of a frame to be executed at time t_M. Most likely some steps need to be taken in order to be able to execute this task at the requested time. For example, we might need to switch on the radio, the frame needs to be loaded into the radio buffer, we might want to set a custom transmit power, the radio needs to be switched to transmit mode until finally it is able to start the transmission of the frame at t_M. Moreover, once the frame transmission is finished we might want to shut down the radio as fast as possible. This task can be described in a chain composed of several commands. In Section 5 this will be explained in more detail.

In order to facilitate the execution of such a sequence of commands we force the use of exactly one master command that has absolute time with respect to the radio time, and slave times that are relative to requested execution time of the master command. Slave commands that are to be executed before the master time have negative relative times while slave commands that need to be executed after the master time have positive relative times. In each chain the master command must be completed on the requested time. Consequently, negative slaves commands must be executed sequentially. It is not allowed for negative slaves to take an alternative path. This is because we want to execute the master command on the requested time. If we would allow alternative paths before the master command, we cannot know when to start the first slave command. However, the MAC designer does not need to worry about those "slave" times. The snapMac Engine will use the *estimateTime2FinalizeCommand* function in order to decide when the first slave command should be executed in order to meet the deadline of the master command. A compiler directive is available to add some time margin (e.g., 30 µs) that will be added to the result of the *estimateTime2FinalizeCommand* function for every negative slave in order to anticipate for unexpected interrupt routines. This way it is possible to achieve the requested execution time of the Master command more precisely. If many external interrupts are expected, we could increase this time margin at the cost of having a (slightly) less energy-efficient solution. If too many external interrupts are expected (e.g., sample the ADC with 100 ksps), we advise to use a separate application processor.

Our chain concept is an alternative for a Finite State Machine. We believe it gives more flexibility to a MAC designer, thanks to the ability to compose, reload or adapt chains at run-time. It also gives the opportunity to accurately plan the execution of commands.

4.5. Command types

We have defined a set of fine-grained commands as a library so that the user level MAC designer can realize any MAC by combining these commands in one or more chains. Every command should be aware that it is part of a chain. Once the *snapEngine* passes the execution token to a command (there can only be one at the time,) it is that command that decides how the chain should proceed.

We have defined different types of commands:

- Continue (C): this is the default type. After execution of a command, the next command in the chain is executed. Commands with command type C can be used anywhere in the chain. All the other commands are not allowed as negative slaves.

- Skip on condition (S): this type will be used for commands that are verifying a condition (e.g., check if a bit is set or not). If the condition is not true, the next command in the chain will be executed. If the condition is true, we will skip the next command in the chain and continue the execution of the chain.

- Skip on event (T): this type will be used for commands that are waiting on a event (i.e., trigger), during a user-defined duration (timer). If the timer expires (i.e., the event did not occur) the next command will be executed. If the event occurs before the timer expires, we will skip the next command and continue the execution of the chain.

The *snapEngine* owns four special commands: GOTO, JUMP, WTRG (Wait Trigger) and STOP. The GOTO command makes it possible to go to another command (designated by an absolute value) in the chain. We typically have a type S or type T command before a GOTO command, because this allows us to take an alternative path in the chain. The JUMP command allows to jump to another command (designated by a relative offset) in the chain. The WTRG command waits for an event (i.e., trigger) from a module. Other commands must be used to check which event it was (e.g., frame that needs to be transmitted, frame received). The STOP command needs to be placed at the end of every chain. With this command, we can reload the chain automatically, keep the chain in memory for future use or destroy the chain. If a STOP is executed, it will sequentially signal all the events (rx frames, cmd reports, etc.) to the user level to minimize the interference from the upper layers during execution of a chain.

In Table 1, we have listed the data plane toolbox commands. For example, the *RxFM* command will check if a 16-bit field at a given offset in a received frame matches with a given reference (ref). A 16-bit mask can be applied to that 16-bit field in order to know if a certain bit is set or not. If it is set (i.e., condition is true), the next command in the chain will be skipped. This generic command is very powerful. We have used it, for example, to check if the ACK request bit is set in a received frame.

In Table 2, we have listed the CC2520 Radio HAL commands. For example, the AutoAck command will configure the radio to send hardware acknowledgments or not. This can command can only be executed if the start state is ST_NO-LPM2 (not in low power mode 2).

Table 1: The commands of the Data Plane Toolbox. For each command the type, the parameters and a short description is given.

Command	Type	Parameters	Short description
RxTo	T	TimeStampT timeout; bool part;	Skip the next command if a (*part* of a) frame was received. Execute next command if *timeout* (To) is reached.
Rx	C	bool part;	Same as RxTo, but here we wait infinitely long. Execute the next command if a (*part* of a) frame was recevied.
TxTo	T	TimeStampT timeout;	Skip the next command if there is a frame posted in the TX buffer. Execute next command if *timeout* is reached.

continued on next page...

Table 1: The commands of the Data Plane Toolbox. For each command the type, the parameters and a short description is given. (continued)

Tx	C	n/a	Same as TxTo, but here we wait infinitely long. Execute next command when there is a frame posted in the TX buffer.
RxLM	S	uint16_t ref;	Skip if length *ref* of the received frame matches, with timeout.
RxFM	S	uint16_t rxOffset, mask, ref;	Skip the next command if a field at offset *rxOffset* with *mask* in the received frame matches with *ref*.
RxTxFM	S	uint16_t rxOffset, txOffset, mask;	Skip the next command if a field at offset *rxOffset* with *mask* in the received frame matches with a field at offset *txOffset* with *mask* in the top frame of the TQ buffer.
TxFM	S	uint16_t txOffset, mask, ref;	Skip the next command if a field at offset *txOffset* with *mask* in the top frame of the TX buffer matches with *ref*.
RxFC	C	uint16_t * dest; uint16_t rxOffset, mask;	16 bit copy from offset *rxOffset* with *mask* in received frame to memory $*dest$.
RxCF	C	uint16_t * src; uint16_t rxOffset, mask;	16 bit copy to offset *rxOffset* with *mask* in received frame from memory $*src$, with mask.
RxS	C	uint16_t signal	Signal the received frame to the upper layer if signal == 1. If signal == 0, drop the received frame and recycle the buffer.

continued on next page...

Table 1: The commands of the Data Plane Toolbox. For each command the type, the parameters and a short description is given. (continued)

TxS	C	uint16_t signal	Signal the transmitted frame to the upper layer if signal == 1. If signal == 0, destroy the transmitted frame and recycle the buffer.

Table 2: The commands of the CC2520 Radio HAL module. For each command the required start state, the stable end state (if applicable) and a short description is given. All commands are type C, expections are indicated.

Command	Required start state	Stable end state	Short description
Off	ST_NO-LPM2	ST_OFF	Turn the radio off.
Sleep	ST_NO-LPM1_2	ST_SLP	Turn the radio in sleep mode.
OnFromOff	ST_OFF	ST_ON	Turn the radio on (from off mode).
OnFromSleep	ST_SLP	ST_ON	Turn the radio on (from sleep mode).
OnFromRF	ST_ALL-RF	ST_ON	Turn the radio on (from RF-on mode).
RX	ST_ON	via sub-chain: ST_RX	Radio in receive mode (requires ON start state)
TX	ST_ON, ST_RX	via sub-chain: ST_RX	Transmit frame.

continued on next page...

Table 2: The commands of the CC2520 Radio HAL module. For each command the required start state, the stable end state (if applicable) and a short description is given. All commands are type C, expections are indicated. (continued)

CcaTxFr	ST_ON, ST_RX	via sub-chain: ST_RX	First do CCA, if OK then transmit frame.
Scan	ST_ON	via sub-chain: ST_SCAN	Let the radio scan (symbol search is disabled)
SwitchChan	ST_NO-LPM1_2	n/a	Change the channel.
LoadFr	ST_NO-LPM1_2	n/a	Load frame in the radio chip.
Prom	ST_NO-LPM1_2	n/a	Promiscous mode.
TxPwr	ST_NO-LPM1_2	n/a	Change the transmit power.
AutoAck	ST_NO-LPM1_2	n/a	Configure the auto ACK.
ScanRate	ST_NO-LPM2	n/a	Configure the scan rate.
RSSI	ST_ALL-RF	n/a	Read the RSSI.
FlushRxB	ST_NO-LPM1_2	n/a	Flush the receive buffer in the radio chip.
FlushTxB	ST_NO-LPM1_2	n/a	Flush the transmit buffer in the radio chip.
FPTH	ST_NO-RX-LPM1_2	n/a	Set the FIFOP threshold.
StartOfFrame	ST_ALL-RF	n/a	Trigger begin of the frame. (Type T)

continued on next page...

Table 2: The commands of the CC2520 Radio HAL module. For each command the required start state, the stable end state (if applicable) and a short description is given. All commands are type C, expections are indicated. (continued)

StartOfFrameInf	ST_ALL-RF	n/a	Trigger begin of the frame, wait infinity.
EndOfFrame	ST_ALL-RF	n/a	Trigger end of the frame. (Type T)
EndOfFrameInf	ST_ALL-RF	n/a	Trigger end of the frame, wait infinity.
CcaControl	ST_NO-LPM1_2	n/a	Configure the CCA parameter.
Cca	ST_ALL-RF	n/a	Trigger the CCA, rising edge or falling edge, with timeout. (Type T)
CcaInf	ST_ALL-RF	n/a	Trigger the CCA, wait infinity.

4.6. The snapMac Engine

The snapMac Engine (so-called *snapEngine*) is the center of our architecture. Every module (both hardware abstraction and toolbox) exposes the *Command* interface to the snapEngine. In this scalable "star topology" it is very easy to add new modules.

A chain is posted from the user level MAC to the *snapEngine* by using this *snapMacControlPlane* interface (see Listing 3).

```
command error_t postCommand(ChainT chainId, SlaveT slaveId,
    CommandT cmdId, ParamT param, StateT blockingState, bool
    eventOnExcec);

command error_t executeAt(ChaintT chainId, TimestampT valid,
    StateT referenceState);

command error_t executeNow(ChainT chainId);

command error_t removeChain(ChaintT chainId, SlaveT slaveId);

command uint16_t listAllCommands(uint8_t typeOfInfo);

event void commandInfo(ChainT chainId, SlaveT slaveId,
    CommandT cmdId, uint16_t error, TimeStampT ts);
```

Listing 3: The snapMacControlPlane interface exposed by the snapMac Engine to the user level MAC (The keywords command and event at the beginning of the functions are nesC related keywords explained in [16], not to be confused with our snapMac Commands and Events)

A user level MAC designer has to choose a unique chain identifier, and post all the commands and associated parameters for this chain one by one using the *postCommand* function. Each command in a chain needs a unique index number (the slaveId). The master command is designated with slaveId zero. All the commands before the master command are called negative slaves, all the commands after the master command are called positive slaves. The slaveId can be used if the user level MAC designer wants to use the GOTO command.

Posting commands from the user level to the kernel level only happens when the user wants to test a new MAC protocol or when the user wants to update the arguments of one or more existing (scheduled) commands.

In case of a new MAC protocol, all the commands must be posted (only once) before the *snapEngine* can execute this new chain. This chain is stored in the *snapEngine*, so that no further command control traffic between user level and kernel level is required after posting a complete chain. Posting (a large number of) commands of a new chain during the execution of an existing chain does not interrupt the execution of the existing chain. To guarantee good system functioning, only one chain can be active at any given time.

In case of updating the arguments of one or more commands, the updates are effective the next time the commands are executed.

We already introduced the subchain concept. In Figure 6, the *TX* command is a subchain with three subcommands. An extra feature of our system is that we can start executing the next command before or during the execution of a (sub)command. When using the *postCommand* function, the MAC designer can specify a so-called "blockingState" for a command. When that is executed and this "blocking state" is reached, we will immediately start the next command (e.g., this can be used to store the timestamp when we start a transmission). The only restriction is that the next command must not change the state of the module that is already executing the previous command (this may be a subcommand). Conversely, the execution of a command A is "full blocking" if it is *not* allowed to start the next command during the execution of command A (this is the default use). It is like a full protection that only command A is allowed until it finishes its execution.

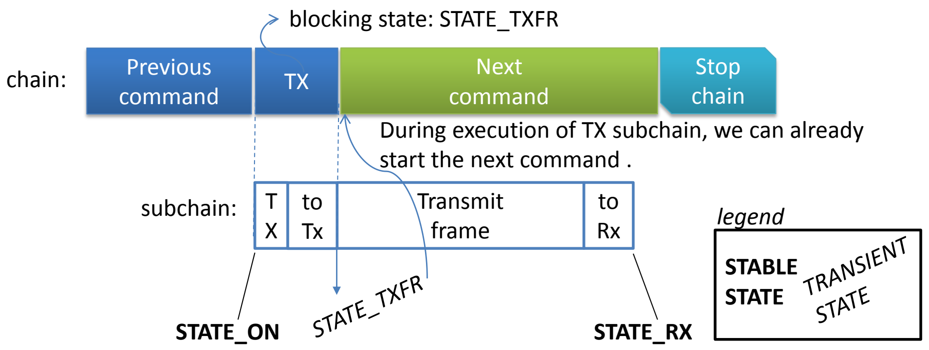


Figure 6: We can start the next command in the chain, based on a "blocking state" in a subchain. In this example, STATE-TXFR was specified as the "blockingState" in the *postCommand* function. So, after the first turnaroundtime, the next command is started in parallel while the current command is transmitting the frame.

Once a chain is completely posted, it can be started in two ways. The *executeAt* function lets the MAC designer plan, at run-time, the time-accurate execution of a chain's master command. The user-defined time t_M is the time when the master command (or a subcommand of the master command) must be valid. For example in Figure 7, it would be the actual "start of transmission" time that the user level MAC designer wants to specify.

Because "transmit frame" is a subcommand of the TX command, the MAC designer can associate the suitable reference state to the user-defined time t_M. We can see that t_M is associated with the transient reference state "STATE-TXFR" because the MAC designer wanted to plan the actual start

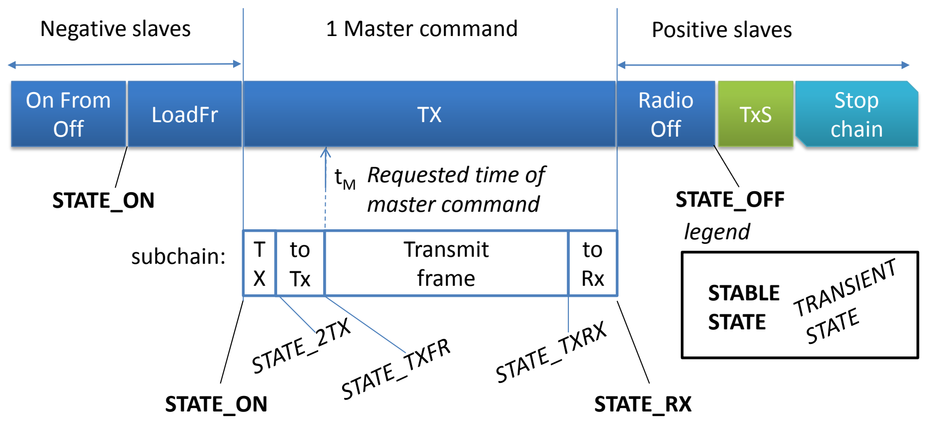


Figure 7: Fine-tune the execution time of a command by associating the given time t_M with a (transient) reference state of a subchain.

of a frame transmission. The *executeNow* function will start the chain as soon as possible, but we recommend to use the *executeAt* function.

The main task of the snapEngine is to make sure that the chains are executed on their requested time. When a chain is posted (or reloaded), the snapEngine scheduler will sort it (earliest deadline first). The snapEngine will handle the chains one by one. The snapEngine dispatcher instructs the suitable module(s), just in time, to execute the commands.

4.7. Transition and Execution Timings of Commands

For every command in the radio HAL module, we first need to know if it is allowed to execute that command given the current state. For example, according to the datasheet of the CC2520 chip it is not allowed to go from STATE_OFF to STATE_SLEEP. We have defined 16 states, so we can use a 16 bit "allowed state mask". Next, we will define the time type (transition, execution, fixed or per byte) and the time needed in sub-symbol ticks (µs in our case). If the snapEngine wants an estimate of the time needed to finalize a command, it will be calculated from stable state to stable state. The example in Table 3 will show the total time needed for transmitting a frame. The TX command is only allowed if the radio state is STATE_ON or STATE_RX. If the allowed state matches with the current state, the command is executed. The next state of this command is STATE_2TX, a transient state. Since each command must end with a stable state, there is a next subcommand listed in the table. The $2TX$ subcommand needs a transition time of 192 µs,

Command	Allowed state mask	Time type	Time [µs]	Next state	Next cmd
TX	ST_ON, ST_RX	Execution	4	ST_2TX	2TX
2TX	ST_2TX	Transition	192	ST_TXFR	TXFR
TXFR	ST_TXFR	Fixed, per byte	160 32	ST_TXRX	TXRX
TXRX	ST_TXRX	Transition	192	ST_RX	NONE

Table 3: Part of the execution and transition time table, to show the time needed for transmitting a frame. The time needed for actual transmission of a frame is given per byte (160 µs for the complete preamble, 32 µs per byte).

after which it is in STATE_TXFR. The next command is $TXFR$. This will transmit the preamble and the complete frame. The preamble (5 bytes) needs a fixed time of 160 µs. Based on the length of the frame, the radio chip will need 32 µs per byte (in the 2.4 GHz band). Once the frame is transmitted, another transition is needed to stable state STATE_RX. This is the turnaround time needed to switch the radio to RX mode. The table can be configured at design-time. If someone wants to port snapMac to a platform with another PHY, it is very easy to change for example the transmit time needed per byte.

4.8. Frame Formatting in User Level and Time-critical Responses in Kernel Level

The IEEE 802.15.4 radio chip allows us to define most of the frame format in software. Figure 8 shows a schematic view of the IEEE 802.15.4 Frame Format.

We did not change the PHY synchronisation header, the length byte in the PHY header and the two Frame Check Sequence bytes in the MAC footer. The CC2520 datasheet states: "*The PHY Service Data Unit contains the MAC Protocol Data Unit (MPDU). It is the MAC layer's responsibility to generate/interpret the MPDU, and CC2520 has built in support for processing of some of the MPDU subfields.*" This is what snapMac offers to the user level MAC designer: the freedom to generate/interpret the MPDU in software and/or using the built in hardware support.

A lot of MAC protocols use time-critical responses to occurring events like for example sending an acknowledgment (ACK) on receipt of a frame.

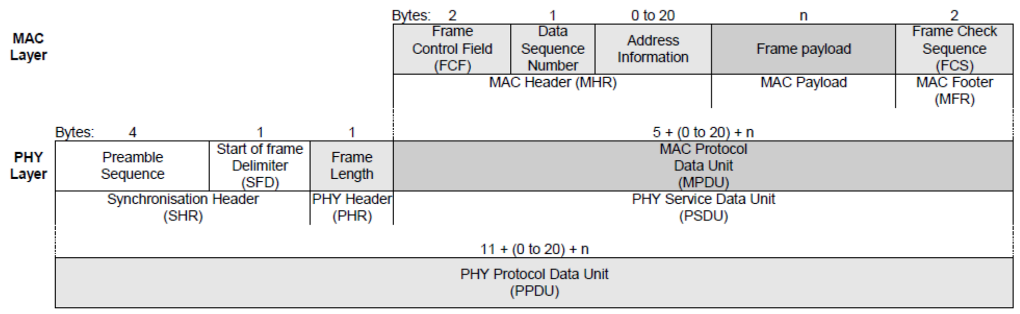


Figure 8: Schematic view of the IEEE 802.15.4 frame format

Passing the received frame to the user level and creating an ACK there will lead to high delays (at least 1 ms). Through the usage of frame templates the user level MAC is able to determine the frame formatting for time-critical responses.

For example, the MAC designer could choose to disable the CC2520 support for generating ACK frames automatically. This way it is up to the MAC designer to decide where to add the ACK request MPDU subfield, and to generate a software ACK frame. The content of this software ACK frame template is also defined by the user level MAC designer and is not pre-defined in the radio driver.

First we will use the *RxFM* command to check if the ACK request MPDU subfield is set. Using a mask and a byte offset, the user level MAC designer can decide which subfield that needs to be checked. Next we need to copy the sequence number of the received frame to the ACK template. This is done with the *RxFC* command. Finally, the template can be loaded in the transmit buffer with the *LoadFr* command. No interaction with the user level MAC is necessary in order to send time-critical responses on incoming frames. Our MAC/PHY abstraction architecture is agnostic to the message format used in the user level MAC. This feature allows for maximal flexibility in frame formats. Radio driver developers also benefit from this approach as they do not need to bother about frame types, therefore resulting in a cleaner (and faster) implementation.

4.9. Conclusion

Our architecture differs significantly in several aspects with respect to current implementations in the WSN domain. The first difference is the snapMAC concept itself: a clean separation between the user level MAC protocol and the kernel level. In the kernel level, we have a central snapMac Engine in combination with toolboxes and hardware abstraction modules. No MAC-specific time-critical functions are present in the kernel. We also introduced a generic *snapMacControlPlane* programming interface (between the MAC control running in user level and the snapMac Engine running in kernel level). This enables posting of time-annoted command chains that contain the MAC protocol logic. The last aspect relates to our requirement that the radio driver must be independent of the frame format in order to maximize MAC design flexibility. Therefore we brought the frame formatting to the user level MAC. Combining all of the above results in reusability of MAC designs in a generic way while guaranteeing the time-critical execution of any MAC.

5. How To Use snapMAC from a user level MAC Perspective?

Here we will show the transmitter command chain needed for asynchronous multi-channel receiver-based communication (in non-beacon mode). This example is one of the new IEEE 802.15.4e solutions for handling the case where channel asymmetry between two devices can happen. In [28, 29], Tytgat demonstrated the feasibility of channel asymmetry with a similar MAC protocol. Each device will select its designated listening channel on which it wants to receive frames. Each transmitter will switch to the designated listening channel of the receiver. The receiver will check if an ACK is requested in the DATA frame, and will switch to the designated listening channel of the originator to send a software ACK (within the specified timing of the standard). This is illustrated in Figure 9.

Suppose we need to transmit a sensor reading from a device, attached to a machine and powered with an energy harvester. This node will sleep until it has harvested enough energy to perform the sensor reading and transmit it to an always-on receiver. The receiver will send a software ACK back (if this is requested by the transmitter) on the listening channel of the transmitter. For each role (transmitter or receiver), we have created two command chains: the init command chain, and the main command chain. During the initialization of the transmitter (A), we let the snapMAC engine execute the command chain that sets the transmit channel to 26, configures the radio not to send hardware ACKs and puts the radio in sleep mode. Next, the main command

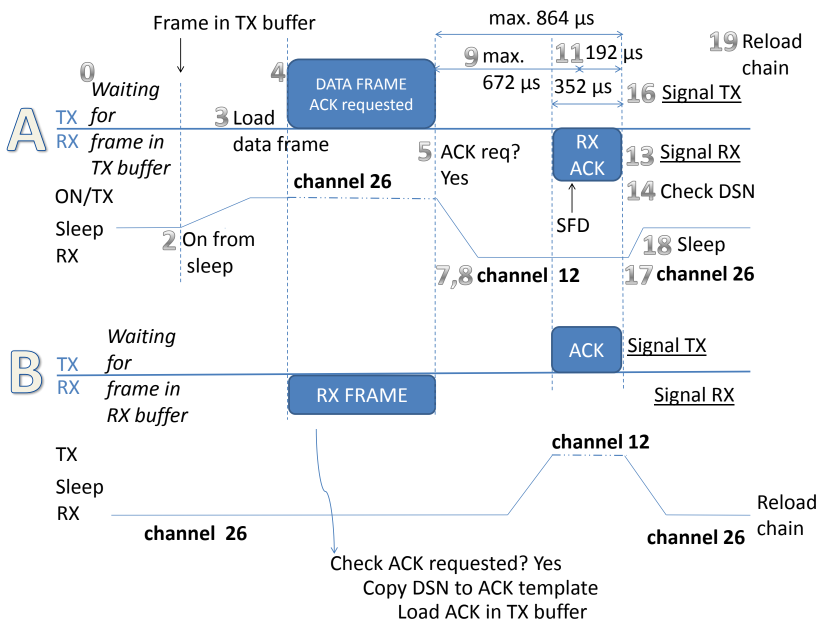


Figure 9: Example of asynchronous multi-channel receiver-based communication. A is the transmitter, B is the receiver. The numbers refer to the *slaveId* identifiers in Figure 10.

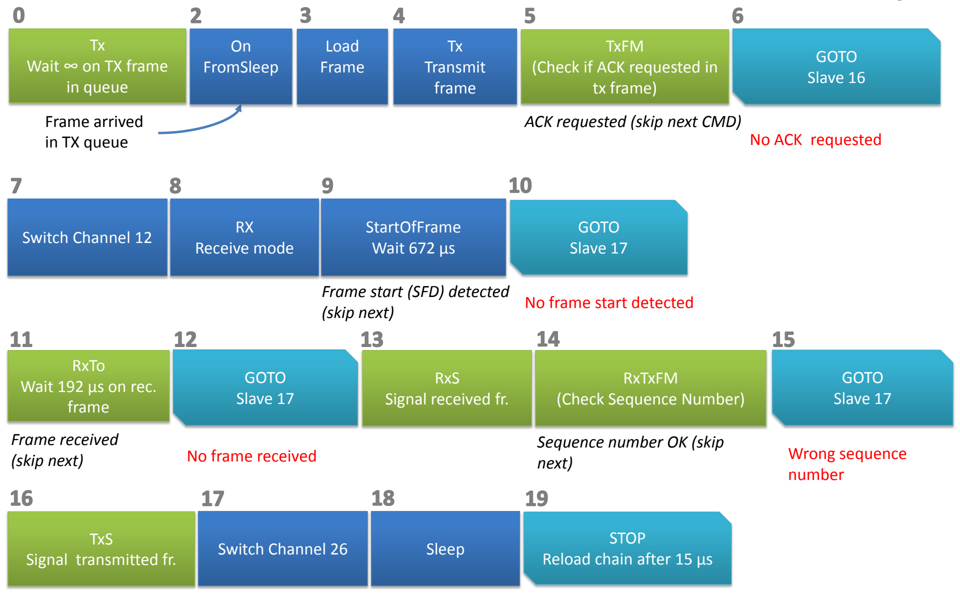

0
Tx Wait ∞ on TX frame in queue
2
On FromSleep
3
Load Frame
4
Tx Transmit frame
5
TxFM (Check if ACK requested in tx frame)
6
GOTO Slave 16
Frame arrived in TX queue
ACK requested (skip next CMD)
No ACK requested
7
Switch Channel 12
8
RX Receive mode
9
StartOfFrame Wait 672 µs
10
GOTO Slave 17
Frame start (SFD) detected (skip next)
No frame start detected
11
RxTo Wait 192 µs on rec. frame
12
GOTO Slave 17
13
RxS Signal received fr.
14
RxTxFM (Check Sequence Number)
15
GOTO Slave 17
Frame received (skip next)
No frame received
Sequence number OK (skip next)
Wrong sequence number
16
TxS Signal transmitted fr.
17
Switch Channel 26
18
Sleep
19
STOP Reload chain after 15 µs

Figure 10: The command chain for the transmitter in this example contains 19 commands. We have not shown the init command chain (configures the radio to use channel 26 and not to send hardware ACKs.)

chain of 19 commands (Figure 10) will be executed (point of time of master command is decided by the user level MAC).

The master command (0) will wait infinitely long for a frame to be transmitted (so we do not skip a command once a frame is ready to be transmitted) Slave 2 will turn on the radio from sleep, slave 3 will load the frame in the radio, and the next command will start the actual transmission on channel 26. Slave 5 will check if the transmitted frame has requested an ACK. If no ACK is requested, the transmission is successful and slave 6 will go to slave 16 in order to signal the transmission to the user level MAC. However, if an ACK is requested, slave 6 will be skipped and we are going to prepare the reception of an ACK on channel 12 (slave 7 and 8). We have chosen to wait for the start of a frame with a timeout of 672 µs . If no frame start is detected before this timeout, command 10 is executed (jump to slave 17). However, if the start of a frame is detected we will skip slave 10 and wait 192 µs for receiving the ACK frame (slave 11). If a complete frame is received, we will signal (slave 13) this to the user level MAC. Next, we will check the sequence number in slave 14. If it is OK, we will skip slave 15 and signal an acknowledged transmission to the MAC. After setting the channel to the designated channel of the receiver, we turn the radio in sleep mode and reload this command chain after 15 µs.

6. Functional Validation and Performance Evaluation

We used our RM-090 nodes to evaluate snapMAC in terms of software ACK timings, throughput, round trip time, beacon interval time accuracy and energy consumption. We have also designed a Low Power Listening MAC. We have created chains, so our experiments also serve as a functional validation of snapMac.

6.1. Experimental Setup

The experimental set-up is shown in Figure 11. Using the Agilent Technologies DS03062A oscilloscope together with a 4.5 Ohm resistor allows us to visualize the current consumption of node *A*. Node *A* is also connected to our Saleae Logic Analyzer. Node *B* and sniffer node *S* are in close vicinity. The sniffer node *S* is connected through USB to our laptop. The sniffed 802.15.4 frames are encapsulated into UDP frames using the ZigBee Encapsulation Protocol (ZEP) and sent to WireShark 1.2.8. The included timestamps are captured on node S, and we have verified the accuracy with a spectrum analyzer.

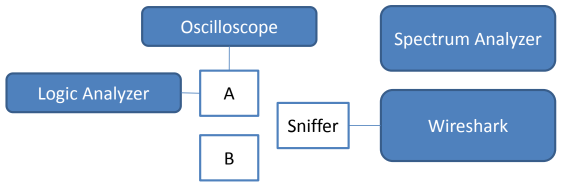


Figure 11: The experimental set-up. Node A, B and sniffer S are RM-090 boards.

6.2. *Software ACK*

Here we want to show that (a) our sofware ACK meets the timing constraints as defined in the 802.15.4 standard and (b) that it is very reliable. As in [30], we refer to the time between transmission of the last byte of a data frame until the reception of the last byte of the complete ACK frame as ACK time. According to the 802.15.4 standard, the maximum ACK time is 864 µs. The ACK time is also illustrated in Fig. 9. We have used the chain as described in the previous section, without the channel switching. We started with a frame length of 9 (minimum allowed data frame length), sent these data frames 500 times at 25 frames per second before we increased the frame length with 6. Our sniffer captured this traffic and based on this trace file we calculated the ACK time and we also measured the reliability.

In Figure 13, we see that snapMac meets the ACK time constraints, as defined in the 802.15.4 standards, starting from a frame length bigger than 15. For all our measurements, the standard deviation was below 2.5 µs and the minimum and maximum ACK times are close to the average. For data frames between 9 and 15 bytes we notice that we are constrained by our hardware platform. Preparing the ACK frame starts immediately after receiving the first 6 bytes of a data frame. This preparation needs some time. For shorter packets (e.g., 9 and 15 bytes), the processing continues after the reception of the last bytes. For longer packets, the preparation of the ACK frame is done before the last byte is received. This is shown in Figure 12

In [30], data frames need to be bigger than 18 bytes in order to meet the 802.15.4 standard time constraints (although they cannot guarantee this because it depends on the overall work load of the node). Another drawback in [30] is that *busy waiting* is used while receiving a frame, which limits

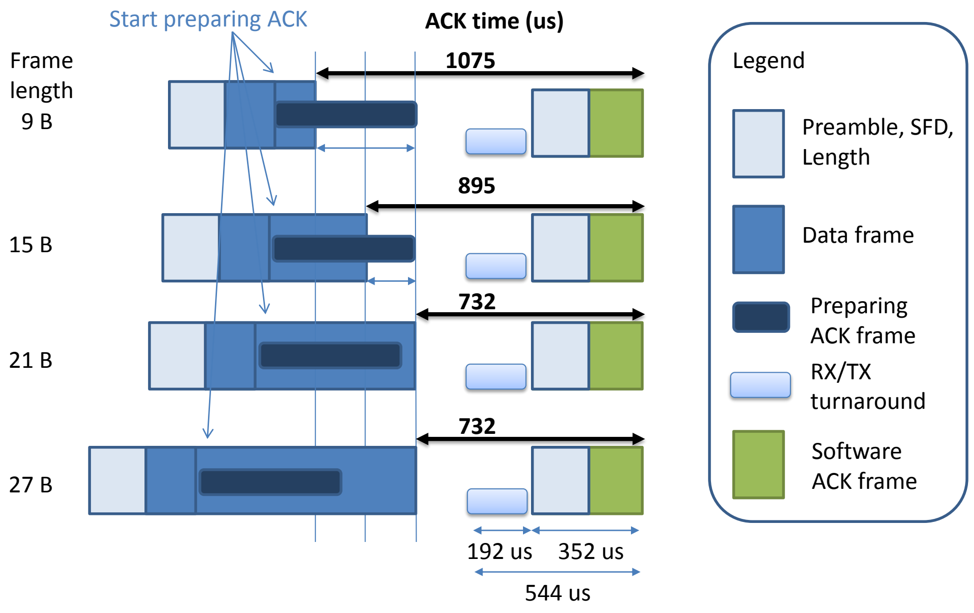


Figure 12: The ACK frame is prepared after receiving 6 bytes of the data frame. For very short frames (smaller than 21 bytes) the preparation of the ACK frame is not finished during the reception of the data frame. For those small packets, the ACK time is a bit higher.

the computational resources for other tasks severely. For small frames, they monitored that 0.08 % of their software-generated ACK frames were lost (vs. 0.0 % for hardware ACKs). Using snapMac 0.00% of the software ACKs were lost during a 1700 sec test (42500 data frames), as illustrated in Figure 14. The bottom line shows the minimum data length (i.e. ACK frames) and the top line shows the increasing/decreasing maximum data length in the WireShark trace file. The extra advantages of software ACKs are the ability to choose (1) a custom transmit power, (2) a custom channel and (3) a custom frame format. This allows to create novel protocols with the ability to test them fast (no driver hacking) and efficiently (e.g., no busy waiting).

6.3. Throughput

We have measured the maximum single-hop throughput by sending maximum sized frames, every 4448 µs, originating from the user level MAC from one node to another. We did not use a backoff scheme and we did not request an ACK. So, the only delay between frames is the turnaround time from tx

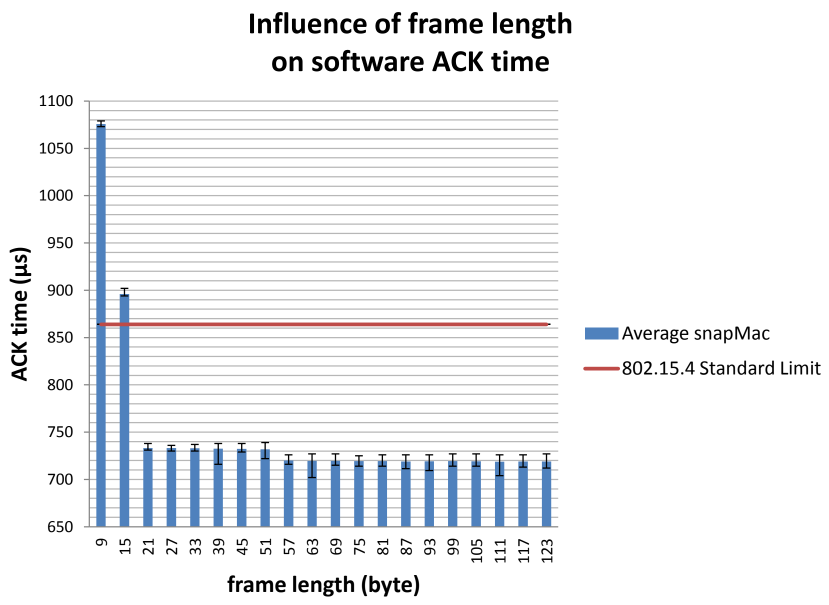


Figure 13: Influence of data frame length on the ACK time.

to rx, loading the frame and the turnaround time from rx to tx.

The number of nodes in a network does not impact the kernel level operation of each individual node. Each individual node has its own engine. This configuration with two nodes is the ideal stress test for the implementation of snapMac because with a larger number of nodes the load of each individual node would be lower. This proofs that the proposed architecture can handle 'heavy' traffic from one transmitter to a receiver.

The initialization chain contains 5 commands, the main chain contains 7 commands. We ran this test during 8 hours and we measured the throughput at 10 random moments (using a sniffer and WireShark to count the number of transmitted frames per second). Sending a maximum sized frame (including the preamble) lasts 4256 µs. The maximum measured throughput is 218 frames per second. For each frame, the maximum data payload size is 125 bytes (everything between the length byte and the Frame Check Sequence). As calculated in Section 2.1 of [31], the theoretical upper bound on the single-hop throughput T_s is 224.82 kbit/s. We achieved 97% (i.e., 218 frames per second times 125 bytes (1 kbit/s) payload is 218.0 kbit/s) of this theoretical upper bound. This is the highest reported throughput. The average time between frames is 332 µs. The fact that we can achieve such a high through-

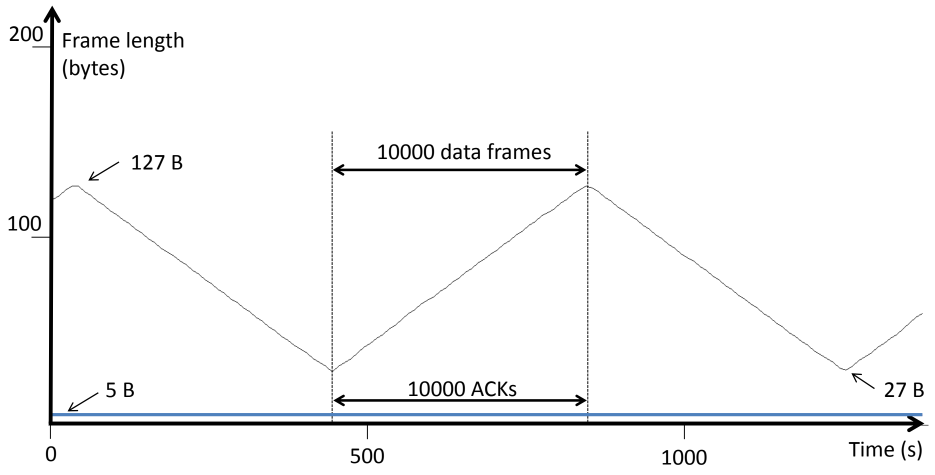


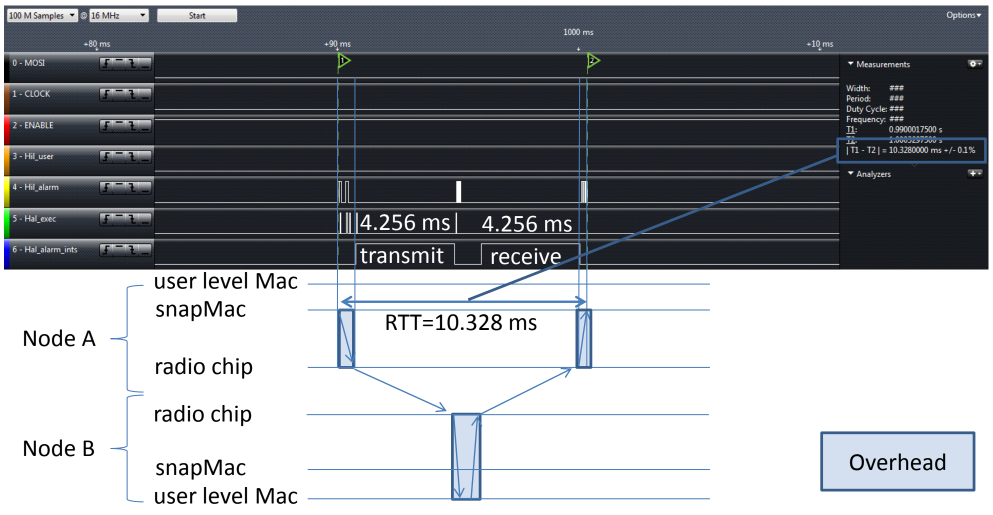
Figure 14: Reliability of software ACKs with data frame lengths increasing/decreasing over time. We have sent 25 frames per second during 1700 seconds. The bottom line shows the length of the captured ACK frames (5 B).

put also means that every single transmission is executed very efficiently. As such we conclude that the overhead for a single transmission is minimal. Minimal overhead also leads to minimal wasted energy consumption. With the standard CC2420 TinyOS driver on a TMoteSky and a Zolertia Z1 mote, the maximum measured throughput is respectively 66 and 89 frames per second. The new CC2420x TinyOS driver is able to transmit respectively 116 and 179 frames per second.

6.4. Round Trip Time

The Round Trip Time (RTT) is the time needed for a message exchange between two nodes. The RTT includes the processing time at both transmitter and receiver side, and the actual transmissions of the messages. For two maximum sized frames, the transmission time is 8.512 ms. We will measure the round trip time in the user level MAC of node *A*. The goal is to determine the overhead of snapMac processing. For this experiment, we have created a chain that puts the radio in receive mode (always-on MAC). Next, we wait on an event (this could be a frame we have to transmit or a frame that is received). We check if a message is in the TX buffer (this check is done in 15 µs). If there is a message ready, we send it. Otherwise, we check if there is a message in the RX buffer (this check is done in 15 µs). At the end of the chain, we signal transmitted or received messages to the user level.

Once the chain is executed, it is automatically reloaded. The initialization chain contains 8 commands, the main chain contains 11 commands.



```
Width:      ###
Period:     ###
Duty Cycle: ###
Frequency:  ###
T1:         0.9900017500 s
T2:         1.0003297500 s
| T1 - T2 | = 10.3280000 ms +/- 0.1%
```

Figure 15: Round Trip Time measured with Logic Analyzer on node A. Round trip time (10.328 ms) starts when the frame is passed to snapMAC. Next it is loaded in the radio and transmitted to node B (4.256 ms), there a new frame is created in the user level of node B, loaded in the radio and transmitted to node A (4.256 ms), and finally made available in snapMac for the user level MAC of node A.

As shown in Figure 15, the RTT is 10.328 ms. The overhead introduced by node A and B together is 1.816 ms. As mentioned in the Related Work section, in [18] the measured RTT was 60 ms for “No CSMA/CA / ACK”, 79 ms for “CSMA/CA / ACK enabled”, and 62 ms for Meshnetics ZigBeeNet MAC”. We conclude that the processing overhead introduced by snapMac is very low.

6.5. Time Accuracy Validations

Here we will show that it is possible to be standard compliant with the IEEE 802.15.4-2006 standard. The Beacon Interval (BI) is dependent on the Beacon Order (BO, valid range 0-14 in beacon-enabled PAN):

$$BI = aBaseSuperframeDuration * 2^{BO}(symbols) \tag{1}$$

We have repeated the same test as done in [27]. In Table 4 the theoretical values of the 802.15.4 BI are compared with the experimental values (for

Table 4: Experimental results of the beacon interval accuracy. The average experimental results are both accurate and precise.

	Theoretic	Experimental			Avg. error		Precision	
	BI	Min	Avg.	Max.	Abs.	Perc.	Std. deviation	
BO	[µs]	[µs]	[µs]	[µs]	[µs]	[%]	[µs]	[ppm]
0	15360	15331	15352	15360	8	0.052	5.4	352
1	30720	30642	30701	30721	19	0.062	12.8	417
2	61440	61342	61418	61442	22	0.036	19.1	311
3	122880	122719	122806	122883	74	0.06	32.1	261

BO = 0, 1, 2, 3). Our oscilloscope could not measure better than 100 µs resolution (given that we have to display minimum 16 ms on the oscilloscope). Therefore, we have used a Saleae logic analyzer to measure the time between each start of a beacon. We took 1 billion samples at 16 MHz (62.5 s). The error of this measurement method is, according to the logic analyzer software, 0.1%. The crystal of the logic analyzer adds ±20ppm to the ±40 ppm crystal of our sensor platform. The total drift of our measurement set-up is 60 ppm. Theoretically, the worst-case drift is 40 µs (on the sensor platform) and 60 µs (in our measurement set-up) per 1 s. In practice, the FLL control loop for frequency stabilization is not working well. The workaround for this hardware constraint is turning off the FLL or implementing a Software FLL [5]. We prefer to add an extra clock source (e.g.,8 MHz, 16 MHz). Our experimental results (shown in Table 4) are both accurate (the average is close to the requested beacon interval, small bias) and precise (the standard deviation is low). Both the average error and the precision do not increase linearly with an increasing BO, indicating that this is a hardware issue and not related to the software. In 802.15.4 networks, a guard time (idle listening) at the receiver is used to compensate clock drift. Clock drift prediction methods can limit this drift uncertainty and shorten the guard times. Such a method is out of scope of this paper, but it is important to know that snapMac supports accurate scheduling.

[5]Errata Number UCS10, MSP430F5437 Device Erratasheet SLAZ287C Revised January 2013

6.6. Energy consumption: sleep - load - transmit - receive ACK - sleep

Here we want to show the energy consumption of a MAC sequence that is often used in duty-cycled MAC protocols. This energy consumption measurement shows that the commands (sleep - load - transmit - receive ACK - sleep) are executed just in time and that the chain handles both cases: receiving an ACK and not receiving an ACK.

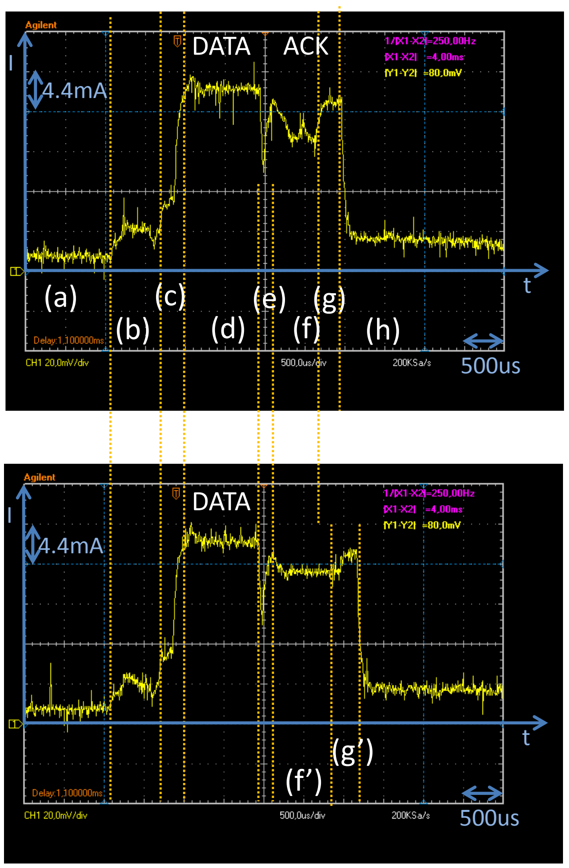


Figure 16: Energy consumption measured on the oscilloscope. In the bottom screenshot, no ACK is received and we can see that the node waits longer: (f) vs. (f').

In Figure 16, we see two screenshots of our oscilloscope connected to node A. On the horizontal axis we see the time (500 µs per division). On the vertical axis we have the current (4.44 mA per division). Node A sleeps until a frame has to be sent (a). The node will go to ON from SLEEP (b), load the frame in the radio chip (c), send it to node B (d), turnaround from tx to rx (e), wait for an ACK (f), inform the user level MAC (g) and go back to sleep and some serial communication (h). In the bottom screenshot, we have disabled node B. We can see that node A is waiting for an ACK for 830 µs (f') and that more energy is consumed when an ACK is not received. This is because searching for the start of a message is more intensive than receiving the message.

6.7. Low Power Listening MAC

Low Power Listening (LPL) is a MAC technique where the transmitter repeats a packet during a certain window, and the receiver periodically samples the channel in order to detect activity (Clear Channel Assessment checks). In our example, node A generates 1 frame per second. Before transmitting, we wait for a clear channel and if the channel is clear we repeat the packet during 206 ms with an inter frame delay of 660 µs. Node B will check for channel activity every 202 ms for 1000 µs. If activity is detected, we will wait for the complete frame in Rx mode, otherwise we go to sleep mode. Once the chain is executed, it is automatically reloaded. The initialization chain contains 9 commands, the main chain contains 31 commands. Compared with MLA [19] components, this command chain includes the commands that create a "channel poller" (sampling the radio channel for activity), an "LPL listener" (adjust the radio's power state based on channel activity) and a "Preamble sender" (sending back to back copies of the packet).

In Figure 17 we see the radio HAL activity (captured with a Logic Analyzer) for node A and node B. First, node A checks if the channel is clear and transmits the frame. This is received by node B. Next, node B echoes the packet when the channel is clear. This experiment confirms that the CCA commands work.

6.8. Real-life deployment stability test

Our snapMAC framework has been used in a musical-theatre production[6] in order to control 10 robots with a TDMA MAC. No stability issues were discovered during this real-life deployment or during our experiments.

For each command exposed by a snapMac module (Data Plane Toolbox, Radio HAL, Antenna Switcher A HAL, Arithmetic Toolbox or a new module), we know exactly how long it takes to execute (see Table 3, these timings

[6]http://www.youtube.com/watch?v=on3fzlzaUjI

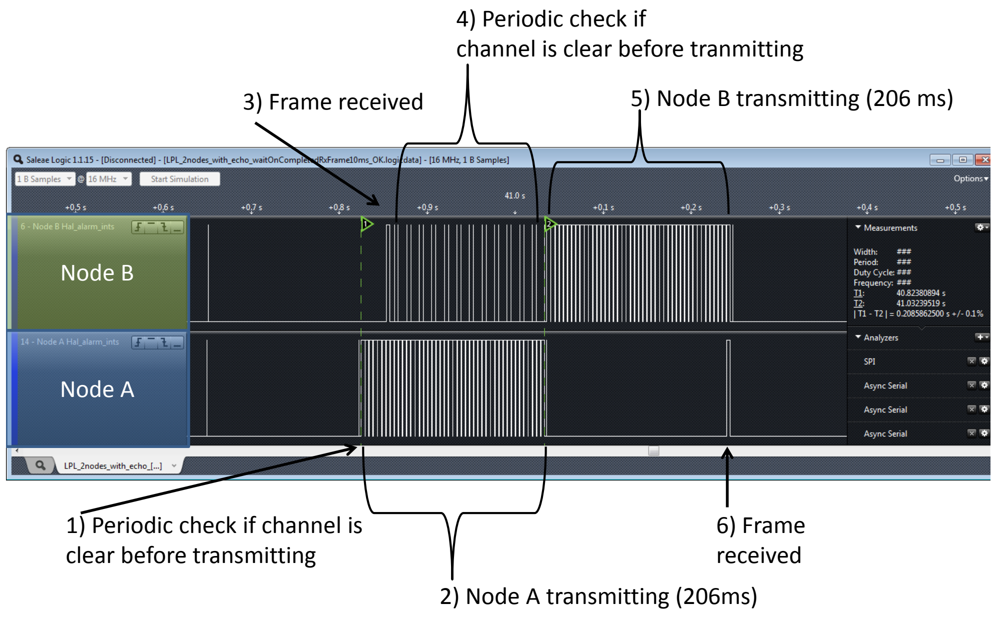

4) Periodic check if channel is clear before tranmitting
3) Frame received
5) Node B transmitting (206 ms)
Node B
Node A
1) Periodic check if channel is clear before transmitting
6) Frame received
2) Node A transmitting (206ms)

Figure 17: Radio HAL activity (captured with a Logic Analyzer) for node A and node B, demonstrating the Low Power Listening MAC.

can be adjusted). Because there is no risk of interrupting the execution from the user level and because each command is handled sequentially, we have created a stable framework.

6.9. Memory Footprint

Here we will show a detailed breakdown of ROM and RAM usage of the snapMac framework without user-defined chains. There is no dynamic overhead because we define at compile-time the maximum number of commands (default: 40) that can be used in user-defined chains. For debugging purposes we also define at compile-time the maximum number of commands (default: 10) for which we want to enable debug information. Both parameters influence the RAM overhead. For each additional command that can be used in user-defined chains the RAM overhead is 28 bytes. For the default maximum of 40 commands, 1120 bytes in RAM is required. For each additional command for which we want to enable debug information the RAM overhead is 14 bytes. For the default 10 commands, 140 bytes in RAM is required. This is only needed during development.

The complete snapMac framework (without user-defined chain(s)), including TinyOS, a microsecond timer, maximum 40 commands and 10 commands with debug information utilizes 11720 bytes of the ROM and 1616 bytes of the RAM. This is 4.6% of the available ROM and 10.1% of the available RAM on our RM-090 node.

Now we have analyzed the RAM usage, we will look at the ROM usage. First we will show the ROM requirements for the different commands:

- The snapEngine commands use 542 bytes in ROM.
- The arithmetic toolbox commands use 416 bytes in ROM.
- The data plane toolbox commands (see Table 1) use 1110 bytes in ROM.
- The CC2520 Radio HAL commands (see Table 2) use 1734 bytes in ROM.

The total ROM requirement for all the commands is 3802 bytes. The snapMac framework without any commands and user-defined chain(s), including TinyOS, a microsecond timer, maximum 0 commands and 0 commands with debug information utilizes (11720-3802) 7918 bytes of the ROM and 326 bytes of the RAM. TinyOS without the snapMac framework uses 1048 bytes of the ROM and 8 bytes of the RAM. We conclude that the overhead of the bare snapMac framework is 6870 bytes in ROM and 318 bytes in RAM. This is the code that can be completely reused. The material-independent commands (snapEngine, arithmetic toolbox and data plane toolbox commands) can also be reused: this adds 2068 bytes in ROM. In total, 8938 bytes in ROM can be reused. This is 84% of the snapMac framework. The material-dependent commands use 16%. This is shown in Fig. 18.

It is easy to disable commands (at compile-time) that are never used in the user-defined chain(s). This way the total ROM requirement can be further reduced.

We were not able to compare our framework with a monolithic implementation (because there is no monolithic implemenation available for our hardware platform). In TinyOS there is a CC2520 driver, using the tasklet mechanism, for the SAM platform. According to Eric Decker “the existing cc2520 driver isn’t very good” and “it is tightly wedded to the sam platform”

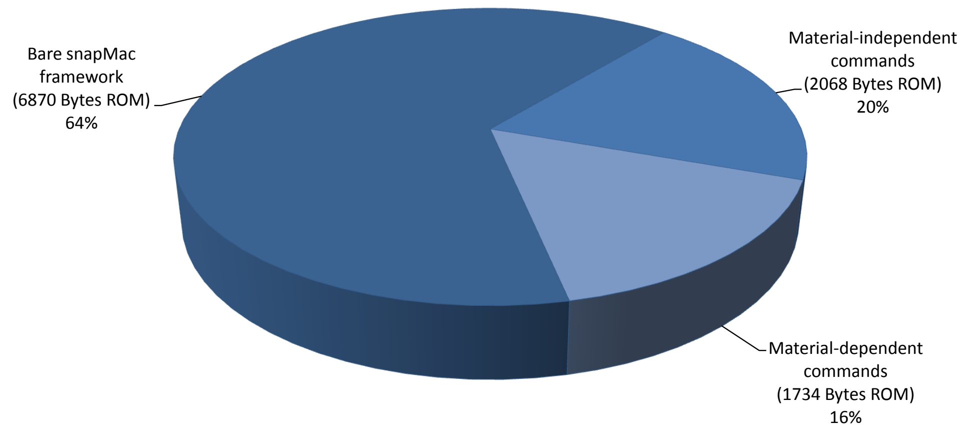


Figure 18: snapMac ROM footprint: the material-dependent commands use 16% of the total snapMac framework.

[7]. Other researchers in our research group have ported the “native” contiki driver to our hardware platform. Next, they will port snapMac to Contiki. At that time, it will be possible to compare the memory footprint of snapMac MAC protocols with “native” protocols.

6.10. Overhead of run-time loading, evaluating and updating of the chain

The one-time overhead of loading the commands of a particular chain depends on the order of the posted commands. When the n commands of a chain are posted in the correct order, the snapMac engine does not need to re-order these commands.

At this moment we have an overhead of approximately 40 µs (16 MHz microcontroller) per executed command (without optimizations) including state machine checking (which could be disabled in deployment).

At any time it is possible to update the parameter of a particular command. The next time the command is executed, the updated parameter will be used.

[7]http://tinyos-help.10906.n7.nabble.com/BlinkToRadio-on-micaz-using-cc2520-not-working-td23577.html October 7 2013

7. Future work: portability of snapMac

In this section we describe the different options of snapMac regarding the portability of the framework and the MAC protocols.

7.1. Framework

Supporting new hardware platforms (i.e., another PHY radio chip) only requires a minimum of changes if the same OS (i.e., TinyOS) is already supported on the new hardware platform. For example, the Zolertia Z1 hardware platform supports TinyOS, but has an older radio chip (CC2420). For this radio chip we need to develop an appropriate radio HAL module in snapMac. The snapEngine and the Toolbox modules remain unchanged. The material-dependant Hardware Abstraction modules (e.g., Radio HAL, Antenna Switcher, etc.) define their own commands and expose their commands via the unaltered Command interface. The commands listed in Table 2 are most likely applicable to a new radio chip, with exception of the *SwitchChan* command if the radio operates on a single channel. For each of these commands, a specific implementation for the hardware chip is required. If the hardware chip offers extra (chip specific) functionality, extra commands can be added to the new Hardware Abstraction module. Our Command interface is generic, so this interface will remain the same on all the supported hardware platforms. The MAC designer is now able to use these new commands in its composition of the command chain(s) at user level.

Our CC2520 Radio HAL module has 1020 lines of code, of which 516 lines of code are specific for the CC2520. Supporting the CC2420 radio chip would require replacing these 516 lines of code with the CC2420 specific instructions.

Supporting another OS (e.g., Contiki OS) requires a port of the snapMac framework. If the new OS is written in C, most of the nesC code can be reused, but one would need to adapt snapMac to the constructs used in the new OS. Once snapMac is ported to a new OS, the user-defined chains can be reused. This would allow us to port existing MAC protocols accross different operating systems. We have not yet completed the port to Contiki OS.

7.2. MAC protocols

With snapMac it will be possible to reuse a MAC protocol chain accross different hardware platforms with the *same* PHY. Many IEEE 802.15.4 compatible radio chips are available. Once the radio HAL module for a new chip is written, all the existing MAC protocol chains can be reused.

Reusing a MAC protocol chain accross different hardware platforms with a *different* PHY is also possible, but the MAC designer should check the capabilities and features of the new PHY. The IEEE 802.15.4 supports different PHYs (frequency, data rate, modulation). Supporting different frequencies and data rates (e.g., 868 MHz, 20 kbps) only requires two updates: (1) change the supported channel list, (2) change the time needed to transmit (this time, as configure in Table 3, relates to the data rate). In [32] an 802.15.4 based MAC is adapted on a powerline communication (PLC) non-wireless medium. This demonstrates the future applicability of our snapMac framework.

8. Conclusions

The major goal of the snapMAC concept is to foster reusability of MACs on different hardware / software / radio platforms.

We have shown that the snapMac architecture supports any time-accurate user level Full Control MAC (1) which communicates through a generic radio interface (2) with a MAC protocol agnostic radio driver (3) that allows generic time-critical interactions (4) while maintaining maximum flexibility and freedom for MAC designers (5). The snapMac architecture changes MAC implementation significantly. First of all, we invented a scheduling (planning) based technique (time-annotated chains) to describe the MAC protocol logic. Second, frame formatting is done in user space in contrast to driver space. Third, a generic MAC protocol agnostic driver can respond instantly on events. Fourth, we have defined two generic MAC protocol agnostic interfaces between the MAC and the radio driver.

This architecture has been implemented and evaluated on the RMoni RM-090 wireless sensor nodes running the TinyOS operating system. We showed that the ACK time is within the 802.15.4 limit for frames bigger than 15 bytes. We also showed that we can reach up to 97% of the theoretical throughput. Our implementation enables energy-efficient execution of duty-cycled MAC designs because we can plan the execution of commands just in time. Although our implementation is oriented towards WSNs, the architecture and the interfaces can be used on PCs with any wireless NIC. In such a scenario the proposed snapMac 'driver' part is the firmware running on the NIC, and any MAC can be implemented on the PC's user level. The only task of the kernel would then be to enable communication between the PC's user level MAC and the NIC firmware.

Concluding we can state that the snapMAC architecture increases reusability and decreases complexity of MAC implementations as well as the underlying radio drivers. Moreover the architecture enables time-accurate execution of radio-related actions and instant reaction on time sensitive events in a generic way.

References

[1] J.-P. Vasseur, A. Dunkels, Interconnecting Smart Objects with IP: The Next Internet, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[2] Libelium sensor applications ranking, http://www.libelium.com/top_50_iot_sensor_applications_ranking/, 2013.

[3] A. Bachir, M. Dohler, T. Watteyne, K. K. Leung, Mac essentials for wireless sensor networks, Communications Surveys & Tutorials, IEEE (2012) 222–248.

[4] P. Huang, L. Xiao, S. Soltani, M. Mutka, N. Xi, The evolution of mac protocols in wireless sensor networks: A survey, Communications Surveys Tutorials, IEEE PP (2012) 1–20.

[5] 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 (WPANs), Technical Report, 2006.

[6] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer, Technical Report, 2012.

[7] Deliverable d1.4 converged architectural reference model for the iot v2.0, http://www.iot-a.eu/public/public-documents/documents-1/1/1/d1.4/atdownload/file, 2013.

[8] P. De Mil, P. Ruckebusch, J. Hoebeke, I. Moerman, P. Demeester, Pluralismac: a generic multi-mac framework for heterogeneous, multiservice

wireless networks, applied to smart containers, EURASIP Journal on Wireless Communications and Networking 2012 (2012) 166.

[9] C. Bormann, Guidance for light-weight implementations of the internet protocol suite, 2013.

[10] Tmote sky sensornode, http://www.crew-project.eu/portal/wilab/sensornode-tmote-sky, 2013.

[11] Zolertia z1 low-power wireless module, http://www.zolertia.com/ti, 2013.

[12] Libelium waspmote, the sensor device for developers, http://www.libelium.com/products/waspmote, 2013.

[13] Linux device drivers, 3rd edition (chapter 7, section 4), http://www.makelinux.net/ldd3/chp-7-sect-4, 2013.

[14] Tinyos, open source os for low-power wireless devices, http://www.tinyos.net/, 2013.

[15] Contiki, open source os for the internet of things, http://www.contiki-os.org/, 2013.

[16] T. Reusing, Comparison of operating systems tinyos and contiki, Sensor Nodes–Operation, Network and Application (SN) 7 (2012).

[17] J. Hauer, TKN15.4: An IEEE 802.15.4 MAC Implementation for TinyOS 2, Technical Report, 2009.

[18] R. Steiner, T. Mück, A. Fröhlich, C-mac: A configurable medium access control protocol for sensor networks, in: Sensors, 2010 IEEE, pp. 845 –848.

[19] K. Klues, G. Hackmann, O. Chipara, C. Lu, A component-based architecture for power-efficient media access control in wireless sensor networks, 2007.

[20] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, P. Steenkiste, Enabling mac protocol implementations on software-defined radios, in: Proceedings of the 6th USENIX symposium on Networked systems design and implementation, NSDI'09, USENIX Association, Berkeley, CA, USA, 2009, pp. 91–105.

[21] T. Schmid, O. Sekkat, M. B. Srivastava, An experimental study of network performance impact of increased latency in software defined radios, in: Proceedings of the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization, WinTECH '07, ACM, New York, NY, USA, 2007, pp. 59–66.

[22] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, P. Mahonen, Decomposable mac framework for highly flexible and adaptable mac realizations, in: New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on, pp. 1 –2.

[23] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, P. Mahonen, A flexible mac development framework for cognitive radio systems, in: Wireless Communications and Networking Conference (WCNC), 2011 IEEE, pp. 156 –161.

[24] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, Wireless mac processors: Programming mac protocols on commodity hardware, in: INFOCOM, 2012 Proceedings IEEE, pp. 1269 –1277.

[25] P. Djukic, P. Mohapatra, Soft-tdmac: A software tdma-based mac over commodity 802.11 hardware, in: INFOCOM 2009, IEEE, pp. 1836 – 1844.

[26] A. Rao, I. Stoica, An overlay mac layer for 802.11 networks, in: Proceedings of the 3rd international conference on Mobile systems, applications, and services, MobiSys '05, ACM, New York, NY, USA, 2005, pp. 135–148.

[27] A. Hernandez, P. Park, IEEE 802.15.4 Implementation based on TKN15.4 using TinyOS, Technical Report, 2011.

[28] L. Tytgat, O. Yaron, I. Moerman, P. Demeester, Energy awareness in self-growing sensor networks, in: Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2012 IEEE 17th International Workshop on, pp. 241–245.

[29] L. Tytgat, O. Yaron, P. I., I. Moerman, P. Demeester, Analysis and experimental verification of frequency based interference avoidance mechanisms in ieee 802.15.4, Submitted to IEEE/ACM Transactions on Networking, under re-review as of Jan 23 (2013).

[30] W.-B. Pottner, S. Schildt, D. Meyer, L. Wolf, Piggy-backing link quality measurements to ieee 802.15.4 acknowledgements, in: Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on, pp. 807 –812.

[31] F. Sterlind, A. Dunkels, Approaching the maximum 802.15.4 multi-hop throughput, in: Proceedings of ACM HotEmNets, ACM, 2008.

[32] C. Chauvenet, B. Tourancheau, D. Genon-Catalot, 802.15.4, a mac layer solution for plc, In AICCSA'10. ACS/IEEE (2010).