Implementation of an IEEE 802.15.4-2006 Protocol Stack on the Texas Instrument CC2430

Jean-François Wauthy
Faculty of Computer Science
FUNDP – The University of Namur
Namur, Belgium
jfw@info.fundp.ac.be

Laurent Schumacher
Faculty of Computer Science
FUNDP – The University of Namur
Namur, Belgium
lsc@info.fundp.ac.be

ABSTRACT

In this paper, we present our open source implementation of the IEEE 805.15.4-2006 protocol stack on the TI CC2430 micro-controller in the framework of e-health services, and offer an in-depth investigation with respect to the difficulties towards such an implementation, the limitations of the micro-controller and the hardware platform, as well as our hand-tuning efforts to maximize performance on this off-the-shelf platform. With our implementation, we report its autonomy, maximum frame rate as well as the challenges when enabling security extensions.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Wireless communication*; C.2.2 [**Computer Communication Networks**]: Network Protocols; C.2.5 [**Computer Communication Networks**]: Local and Wide-Area Networks; C.2.6 [**Computer Communication Networks**]: Internetworking—*Standards (e.g., TCP/IP)*; C.4 [**Performance of Systems**]; C.5.3 [**Computer System Implementation**]: Microcomputers—*Microprocessors*

General Terms

Design, Performance, Standardization

Keywords

Implementation, IEEE 802.15.4-2006, Wireless Sensors, Energy Consumption

1. INTRODUCTION

The main focus of the WALIBI project (see administrative details in Section 8) is to monitor a patient with the help of many sensors (surveying body temperature, heart beat, blood pressure, etc) located on his/her body. All the sensors are connected to a node that sends data from sensors to a central device. For the project, we chose to use the IEEE 802.15.4 technology mainly because of its long autonomy and its use of the 2.4 GHz ISM band. Since we wanted to be able to eventually modify the behavior of the radio protocols, we looked for an open-source implementation that could run on off-the-shelf hardware.

The open-source implementations that we identified at the time (beginning of 2008) were : TinyOS [15], MeshNetics' ZigBit modules [6] and Sensinode's NanoStack [13].

TinyOS had limited support of IEEE 802.15.4, it only supported sending and receiving radio frames without handling the network management or providing support for the beacon mode. The OpenZB [11] project was trying to provide these functionalities but supported only TinyOS 1, the deprecated MicaZ board [7] and it was only starting to port the code for the TelosB board [14].

MeshNetics' hardware is based on an ATmega 1281v microcontroller and an Atmel AT86RF230 radio transceiver which are not really off-the-shelf hardware, also their IEEE 802.15.4 implementation was not entirely open-source, some libraries were only available as binaries.

Sensinode's implementation was entirely open-source and available from SourceForge under the GPL license and was advertised as running on both TI MSP430 with a CC2420 radio controller (which is the same as the one on the popular TelosB [14] board) and CC2430 micro-controller.

Later after the start of the project, the Contiki [2] operating system implemented support for IEEE 802.15.4 and the TI CC2430 but, like TinyOS, only sending and receiving frames are available without networking management and it was too late for us to start over by implementing support for these functionalities in Contiki.

We then chose to purchase a TI CC2430 based development kit from Sensinode in order to have supported hardware and an open-source IEEE 802.15.4 stack. However, we quickly found out (and was confirmed by Sensinode) that the beacon mode was only supported on the so-called Micro hardware line (based on TI MSP430 and a TI CC2420) that was being deprecated in favor of the new Nano hardware line although they advertised it was supported. It was then decided to start writing our own implementation of the IEEE 802.15.4-2006 standard on the TI CC2430 hardware previously bought. We could not start from TinyOS or Contiki because they both lacked support for the TI CC2430 micro-controller and its development toolchain.

To the best of our knowledge, there is no other open-source implementation of the IEEE 802.15.4-2006 specifi-

PE-WASUN'10, October 17–18, 2010, Bodrum, Turkey.

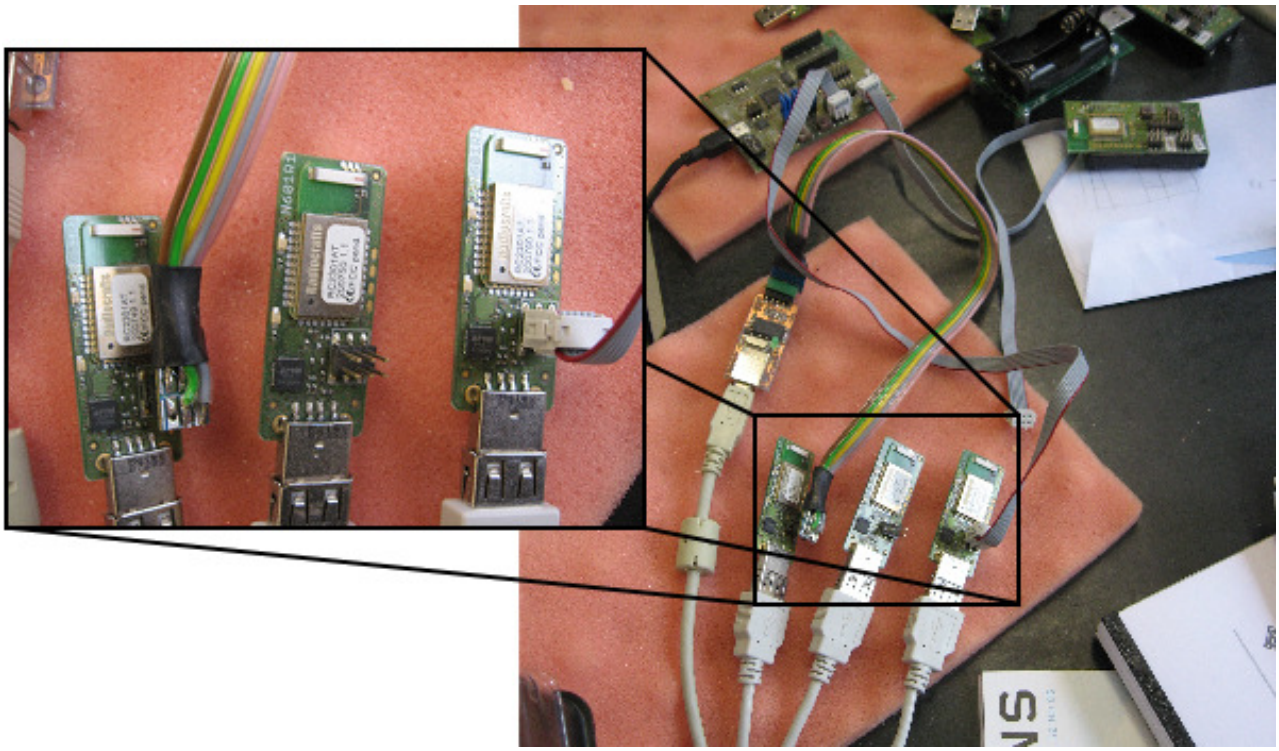

Figure 1: Picture of the development set-up with a close-up showing the coordinator node (left), the sender node (right) and the node capturing the radio traffic for analysis (middle). The programming and debugging boards are also visible above the nodes.

cations supporting beacon mode, associations, and off-the-shelf hardware.

This paper presents our implementation of the 2006 version of the IEEE 802.15.4 specifications as well as the issues and limitations encountered with the hardware during its development. Section 2 presents the hardware, Section 3 describes our implementation of the IEEE 802.15.4-2006 specifications, Section 4 details the optimizations that were made during the development of the stack, Section 5 discusses the main issues that arose during the development, Section 6 quickly presents the license under which our implementation will be released and finally Section 7 concludes and discusses future development.

2. HARDWARE

The TI CC2430 [1] bundles a 32 MHz optimized and enhanced Intel 8051 micro-controller and a TI CC2420 radio controller on a single chip. The CC2430 can hold up to 128 KB of non-volatile program memory (when using memory banking) and 8 KB of data memory. It features, among others, 5 direct memory access (DMA) channels, 8 ADC channels, an AES co-processor, 4 timers whose 1 high definition "MAC" timer, two programmable UARTs, ...

The unusual thing when working with the CC2430 is its memory model inherited from the Intel 8051 architecture and based on an Harvard architecture. The CC2430 CPU has four different memory spaces: the code read-only memory that contains the program itself, the internal data memory quickly accessible but limited to 256 bytes, the XDATA memory whose access is slower than data memory but can address up to 65 KB and the Special Function Register (SFR) memory for direct access to the registers controlling the CPU and the peripherals.

One major drawback of the CC2430 is its very limited debugging capability. It only offers a very limited low level access through the two-wire debug interface. It allows pausing the micro-controller, reading some status bytes and executing the microcode step-by-step but all this is not very practical when developing and debugging a real-time system.

Also the CC2430 does not offer any hardware helpers for implementing dynamic memory allocation or some kind of protected kernel mode. Therefore, dynamic memory allocation support has to be implemented from scratch and its usage might introduce heap fragmentation reducing the amount of memory actually usable by the application, as discussed further in Section 3.3.

3. IMPLEMENTATION

Our implementation of the IEEE 802.15.4-2006 specifications supports beacon mode and security extensions but not the Guaranteed Time Slot (GTS) scheme. It also includes code for the basic functions required to run a coordinator node.

Our code is written in C and is compiled with the Small Device C Compiler (SDCC) [12].

3.1 Layers

As it is usually the case in networking stacks, our implementation of the IEEE 802.15.4-2006 standard can be divided in layers where a layer depends on the ones below it.

These layers are presented on Figure 2 from the bottom to the top.

Hardware layer

The hardware layer provides the functionalities enabling the other layers to use the hardware itself. This includes setting the hardware in a running and usable state (like initialization of the CPU at start up), easily accessing and configuring the Direct Memory Access (DMA) channels, writing data into the UART ports or formatting debug statements.

Physical layer

The physical layer (PHY) provides access to the radio controller and enables other layers to send and receive data over the air. This layer implements the PHY data (PD-DATA) and the physical layer management entity (PLME) interfaces of [5].

The PD-DATA service allows the next higher layer to directly send frames over the air. And the PLME provides the PHY management service to the next higher layer. Its functions can be used to configure the PHY layer through the PHY PAN information base (PIB), perform energy detection or clear channel assessment (CCA).

Data link layer

The data link layer or Media Access Control (MAC) layer provides functions enabling the application layer to organize a network and to send and receive data in a orderly manner in order to avoid conflicts or starvation on the media. This layer implements the MAC common part sublayer (MCPS) and MAC layer management entity (MLME) interfaces of [5].

The MAC data service is accessed through the MCPS interface. The MCPS provides functions to send data with the correct radio channel access method (CSMA-CA or slotted CSMA-CA) and to remove a pending transaction from the queue.

The MLME provides the IEEE 802.15.4 MAC management service to the higher layer. The MLME offers functions to configure the MAC layer through the MAC PIB, to manage the node in the personal area network (PAN), and to alert the next higher layer of events in the MAC (such as loss of beacon synchronization or detection of PAN identifier conflict).

Application layer

The application layer is where the end-user application is located. It can include support for the basic functionalities of a coordinator, so the user does not have to implement them him-/herself.

3.2 One main loop vs. multitasks

First, the whole code of every layer was running inside one main loop in one stack space that processed the incoming frames and allowed frames to be sent directly, sometimes, from inside interrupt service routines. This worked fine with simple use cases like association, disassociation or sending small amounts of data at low rates but when trying to receive data at an higher rate, stack overflow errors occurred on the node running the coordinator code. The transmitted frames are 60-byte long (47 bytes of payload and 13 bytes of headers with no security extension and only short addresses used) and sent at rate of 128 Hz.

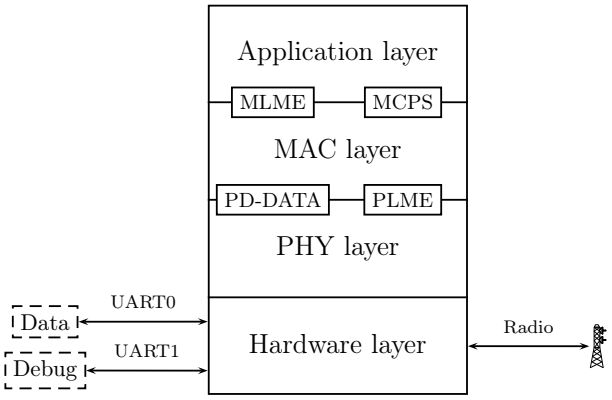

Figure 2: Layers of the IEEE 802.15.4 implementation

Due to these stack overflows, we later started basing our code on FreeRTOS [3], a mini real time operating system (RTOS) providing basic tasks, queues and synchronization mechanisms with support for preemption. However, tasks context switching requires copying the stack of the current task (located in DATA memory) in XDATA memory and copying back the stack image of the new task from XDATA memory into the DATA memory. Now that each task has its own stack space, no more stack overflow happens during high rate reception.

However, for incrementing its internal clock, the kernel requires a timer to generate a regular tick. This combined with context switching introduces latency and overhead in the firmware.

The use of different tasks also complicates the way functions are invoked between tasks contexts (i.e. the application task/layer wants to send a data frame which is managed by the MAC task/layer). To handle this problem, a queue storing pointers to structures containing an identifier of the function to call and its parameters is associated with each task.

We use three to four tasks in our implementation (depending whether coordinator support is activated): the transmission task, the MAC task, the application task and the optional beacon transmission task. To allow them to communicate with other, three message queues interconnecting the tasks are used. The tasks and queues are represented on Figure 3.

3.3 Memory allocation

As previously pointed out, a dynamic memory allocation mechanism has to be completely implemented from scratch on the TI CC2430 but the SDCC compiler implements one and provides the usual functions used in dynamic allocation (malloc, calloc and free). However, this code is not optimal as it has to search through an organized chunk of memory for a large enough space each time a new allocation is required.

A trade-off is to pre-allocate statically a pool of structures of a given size in memory (simply as a C array) and to provide functions to try obtaining a free member of this pool and freeing it once it is not needed anymore. This increases the code size and needs to be finely tuned to mitigate the possible waste of memory due to the unused, but allocated, pool members.

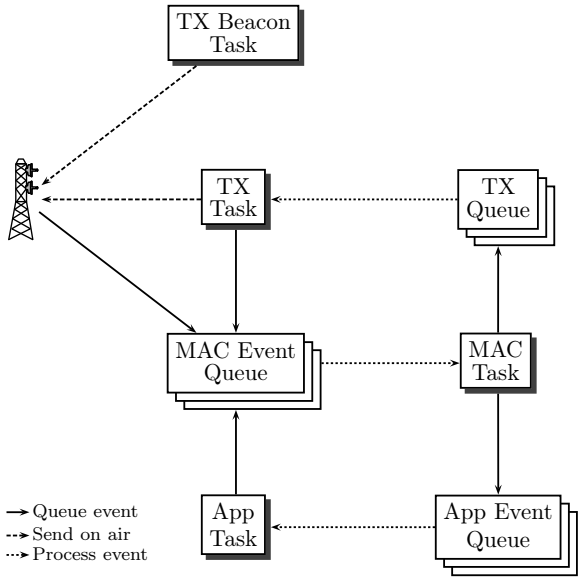

Figure 3: Interactions between tasks and queues

Pre-allocation is used in our implementation for structures with a fixed sized that are often allocated and relatively quickly freed such as structures holding radio frames.

3.4 Debugging

When developing a software, debugging support is very important to ease tracking down bugs and optimize code. Usually developers have access to backtraces allowing them to understand the call path that led to an error, they can also change the value of variables while debugging a running code or place breakpoints that stop the execution of the program when reached.

The debugging support on the TI CC2430 is scarce. There is no Joint Test Action Group (JTAG) connector but a 2-wire interface to an on-chip debug module is available. With an appropriate module and software, it is possible to use the NoICE [10] debugger to debug a program running on the chip at an assembler code level. While this assembler level debugging can be helpful, it is not very suited to understand the call path or to trace it back.

When possible, using one of the UART outputs enables some sort of basic text console where short status messages can be printed. This requires to be able to modify the code to add the debug statements and to recompile the firmware afterwards. It also enlarges the code size, might slow down the execution or even fail to output anything on the UART due to an error (i.e. infinite loop or a blocking call inside an interrupt service routine with the highest priority). Different kinds of debug statements in various subsystems of the stack have been added during development and can be activated using conditional compilation flags.

To help debugging, we also wrote a small firmware for the TI CC2430 that sends the received radio frames to a computer where they can be analyzed with a network traffic analyzer such as Wireshark [16]. The Wireshark dissector for IEEE 802.15.4 frames has also been extended to support the 2006 format of the encrypted frames. These modifications have been submitted to the developers of Wireshark and accepted for integration in the future versions of the software.

4. OPTIMIZATIONS

4.1 DMA for radio

A DMA controller enables copying data within the XDATA address space without using CPU cycles. Since most of the SFR registers are mapped into XDATA memory, it can read from or write data to any peripherals relieving the CPU of handling data movement operations. The controller manages several channels allowing multiple transfers to occur in parallel, these transfers can be also automatically triggered by a pre-configured event (new frame received, transmission completed on an UART, ...). Once the transfer is finished, an interrupt is raised to let the CPU know about it.

Two DMA channels, one for reception and one for transmission, are used to optimize memory transfer between the radio buffer and the micro-controller memory. This enables copying newly received frames or frames that are about to be sent between the buffer of the radio controller and the memory without using the CPU of the micro-controller. The CPU freed from copying the data, it can then execute other instructions in the meantime such as scheduling an other task or processing other requests.

4.2 Conditional compilation

The available flash memory of the micro-controller is limited, therefore in order to reduce the code size of the IEEE 802.15.4 stack leaving more space for the application code, extensive conditional compilation of optional functionalities (FFD-only functions, coordinator support, security, ...) is used. This way, we currently obtain a binary image of 48 kbytes for the coordinator and 38 kbytes for the node compiled without the security extensions.

4.3 Switching off the radio

One important feature for a standalone sensor is its autonomy. To understand when we can power off some component to reduce power consumption, Figure 4 details this structure of an IEEE 802.15.4 superframe. An IEEE 802.15.4 superframe is limited by the transmission of a beacon frame by the network coordinator. It can be divided into an active portion and an inactive portion. The Beacon Interval (BI) is determined by the value of the Beacon Order (BO) parameter selected by the coordinator. The coordinator also selects the value of the Superframe Order (SO) parameter that determines the length of the active portion of the superframe also known as the Superframe Duration (SD). This portion is divided into 16 equally spaced slots.

The active portion is composed of 3 parts: a beacon, a Contention Access Period (CAP) and a Contention Free Period (CFP). The beacon is transmitted at the beginning of slot 0 and the CAP starts immediately after the beacon. The CFP, if present, is composed of the Guaranteed Time Slots (GTSs) and is located immediately after the CAP up to the end of the active portion of the superframe. The network coordinator may allocate a maximum of 7 GTSs as long as the CAP is at least 440-symbol long. Figure 4 contains 2 GTSs spanning over 5 superframe slots.

To reduce the node current consumption, we switch off the radio during the inactive period of the superframe because with the radio left on even without receiving any frame the

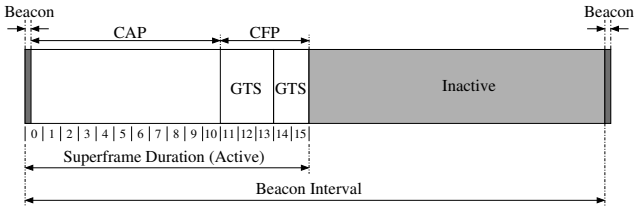

Figure 4: IEEE 802.15.4 Superframe structure representing the Contention Access Period (CAP) and Contention Free Period (CFP) and Guaranteed Time Slots (GTSs)

CC2430 draws around 28 mA. With the radio shut down, the current consumption goes down to an estimated 11 mA. Transmitting a frame requires a little bit more current and is estimated around 30 mA. These values are based on the datasheet electrical specifications and considering an additional consumption for the activated peripherals.

To evaluate the autonomy with these estimated current consumptions, we divide the beacon interval in three parts and evaluate the consumption of each of them. These parts are the beacon duration during which the beacon is either sent or received, the active portion and the inactive portion. Their current consumption can then be divided by the duration of the beacon interval to obtain the mean current consumption of the node.

Since the bitrate of IEEE 802.15.4 in the 2.4 GHz ISM band is 250 kbps, the beacon duration can be expressed as (in μs):

$$\begin{aligned} BeaconDuration &= \frac{BeaconLength \times 8 \times 1000}{250} \\ &= BeaconLength \times 32 \end{aligned}$$

The BI is defined in the IEEE 802.15.4 specifications as the duration of a superframe when the SO is equal to 0 multiplied by 2^{BO} where BO is the beacon order. The BI can be derived as:

$$\begin{aligned} BeaconInterval &= aBaseSuperframeDuration \times 2^{BO} \\ &= aBaseSlotDuration \\ &\quad \times aNumSuperframeSlots \times 2^{BO} \\ &= 60 \times 16\,sym. \times 2^{BO} \\ &= 60 \times (16 \times 16\,\mu s) \times 2^{BO} \\ &= 2^{BO} \times 15{,}360\,\mu s \end{aligned}$$

where *aBaseSuperframeDuration*, *aBaseSlotDuration*, *aNumSuperframeSlots* are constants defined in [5] and the symbol duration is 16 μs since the CC2430 makes use of the 2.4 GHz version of IEEE 802.15.4.

The duration of the active portion of the BI is defined as:

$$\begin{aligned} ActiveDuration &= aBaseSuperframeDuration \times 2^{SO} \\ &= 2^{SO} \times 15{,}360\,\mu s \end{aligned}$$

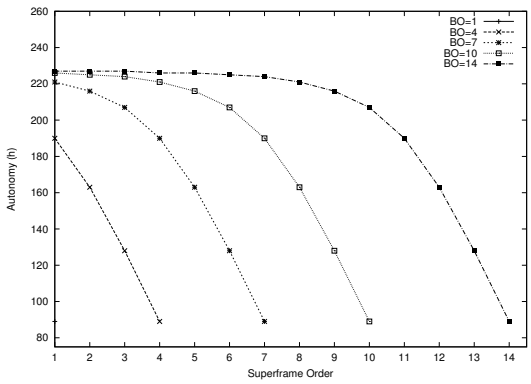

Figure 5: Mean autonomy of an idle coordinator in beacon mode

The total current consumption can be then expressed as:

$$\begin{aligned} Consumption = (&BeaconDuration \\ &\times BeaconConsumption) \\ &+ [(ActiveDuration - BeaconDuration) \\ &\times ActiveConsumption] \\ &+ [(BeaconInterval - ActiveDuration) \\ &\times IdleConsumption] \end{aligned}$$

We can then obtain the mean current consumption:

$$MeanConsumption = \frac{Consumption}{BeaconInterval}$$

Figure 5 shows the mean autonomy of coordinator running on 2,500 mAh batteries and sending a 25-byte long beacon which corresponds to a standard beacon with a small payload and eventually basic security options. Several BOs are considered and the mean autonomy is shown in function of the SO. The decrease is due to the inactive period getting smaller as the SO is nearer to the BO since the coordinator radio must always be on in a beacon enabled network.

The mean autonomy of a node tracking its coordinator's 25-byte long beacon with the same type of batteries as the coordinator is almost constant around 227 hours because the radio is even shut down during the active period of the superframe since the node only needs to have it on to track the beacon and eventually send frames.

The TI CC2430 also features low power consumption modes, if we only consider the less power saving mode, the consumption of the micro-controller can be estimated to 200 μA (according to the manufacturer's specifications). Using this low power mode when the radio is not needed, we can reach an estimated autonomy of around 12,000 hours with the same parameters. Using the other power saving modes with a long inactive period can achieve even longer autonomy with a node idle most of the time.

In our implementation, we support switching off the radio during inactive periods but we did not add support for these low power modes because of the scenario considered. In our project, data frames are sent all the time and since the BO and SO parameters are the same, there is no inactive period in the superframe.

5. ISSUES

During the development of our implementation of the IEEE 802.15.4-2006 specifications, several issues were encountered. The two most important ones were related to the security extensions of the standard and to the support of high rate transmissions required by Walibi.

5.1 Security extensions

The IEEE 802.15.4-2006 MAC security extensions support data confidentiality, data authenticity and replay protection based on an extension of the counter with cipher block chaining message authentication code (CBC-MAC) mode of operation also known as CCM. This extension is named CCM* and is described in [5]. While enabling the security extensions of IEEE 802.15.4, we faced several issues.

First of all, the IEEE 802.15.4 specifications of the security extensions do not detail how to distribute the keys in the network and leave that uneasy task to an other layer. Even storing these keys in the nodes makes use of complicated (for small micro-controllers) data structures; and each node in the network can have several keys depending on the frame type and security level.

Moreover, the activation of the security extensions obviously complicates the processing of incoming and outgoing frames, effectively reducing the transmission rate due to the various checks, key lookups and modifications of the frames that need to take place.

Finally, the IEEE 802.15.4 (either the 2003 or 2006 version) standard uses the little-endian byte order and therefore so does the TI CC2430. However, the security extensions of IEEE 802.15.4 are based on the big-endian byte order which complicates their implementation and prevents the use of DMA on large memory block alternatively it requires to prepare the block in taking into account this different endianness which complicates and enlarges the code.

Our implementation of the CCM* algorithm has nevertheless been validated using CCM test vectors from the National Institute of Standards and Technology (NIST) [8, 9] and from the IEEE 802.15.4 specifications.

5.2 High rate transmission

In our project (see Section 8), we want to transmit electroencephalography (EEG) data. The medical partners of the project agreed on a 128 Hz 11-bit signed sampling of the 20 electrodes. This leads to the following bit rate where headers are the IEEE 802.15.4 headers of the data frame, samples are the sampled data prefixed with a 5-bit sensor identifier that are sent and overhead is the number of additional bytes used to store the sequence number, a checksum and the length of the sampled data:

$$\begin{aligned} Bit\ rate &= SamplingFrequency \\ &\quad \times \Big[\underbrace{(13 \times 8)}_{headers} + \underbrace{20 \times (11+5)}_{samples} + \underbrace{(7 \times 8)}_{overhead} \Big] \\ &= 128 \times (104 + 320 + 56) \\ &= 61{,}440\ bps \end{aligned}$$

This bit rate is well within the theoretical limits of IEEE 802.15.4 but even when sending two measurements in an IEEE 802.15.4 radio frame and intensively using DMA to transfer data from the UART (from which the data is acquired) and to the radio and vice-versa we could only achieve 100 Hz sampling (which corresponds to 48,000 bps) on the sender node with the multitasks implementation whereas the mono-task supported up to 128 Hz for a period of time, until stack overflow occurs. This is due to the overhead introduced by the multitask implementation where messages are intensively exchanged between tasks. However, the receiver node could handle receiving data sampled at 128 Hz when running either the multitasked code or the basic mono-task code.

6. LICENSING

We plan to release our implementation of the IEEE 802.15.4-2006 specifications under a modified version of the GPL that would prevent commercial usage of the stack while keeping it under the GPL for other usages.

The source of the stack is available through our Git [4] repository. It can be browsed with gitweb at the following address: `http://git.infonet.fundp.ac.be` and cloned using `http://git.infonet.fundp.ac.be/git/picostack` as source repository.

7. CONCLUSION

After detailing the reasons that led us to start writing our own implementation of the IEEE 802.15.4-2006 standard, we presented the TI CC2430 micro-controller and how our implementation is structured and works. Then we discussed the optimizations that we integrated to improve its performance and autonomy. We evaluated this autonomy in function of the BO and SO chosen for the coordinator of the network. Finally, we presented the issues we encountered during the development of the security extensions and producing high rate transmissions.

Future work includes reducing the overhead of the communication mechanism between tasks to ultimately improve the tranmission rate performance and adding support for GTS reservation.

8. ACKNOWLEDGMENTS

The work presented in this paper has been realized within the Wireless Acquisition and LInk for Body Information (WALIBI) research project, a WIST2 project funded by the Walloon region (Grant 616 449).

9. REFERENCES

[1] Texas Instruments CC2430, System-on-Chip Solution for 2.4 GHz IEEE 802.15.4 / ZigBee (TM). `http://focus.ti.com/docs/prod/folders/print/cc2430.html`.

[2] The Contiki Operating System. `http://www.sics.se/contiki/`.

[3] FreeRTOS. `http://www.freertos.org/`.

[4] Git – The fast version control system. `http://www.git-scm.org/`.

[5] The Institute of Electrical and Electronics Engineers. *IEEE 802.15.4-2006: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, June 2006.

[6] MeshNetics. `http://www.meshnetics.com`.

[7] MicaZ sensor board. `http://www.xbow.com/Products/productsdetails.aspx?sid=101`.

[8] National Institute of Standards and Technology. http://www.nist.gov/.
[9] NIST – Cryptographic Algorithm Validation Program. http://csrc.nist.gov/groups/STM/cavp/index.html#07.
[10] NoICE Debugger. http://www.noicedebugger.com/.
[11] Open-ZB. http://www.open-zb.net.
[12] SDCC - Small Device C Compiler. http://sdcc.sourceforge.net/.
[13] Sensinode. http://www.sensinode.com/.
[14] TelosB sensor board. http://www.xbow.com/Products/productdetails.aspx?sid=252.
[15] TinyOS. http://www.tinyos.net/.
[16] Wireshark. http://www.wireshark.org/.