Contiki80211: An IEEE 802.11 Radio Link Layer for the Contiki OS

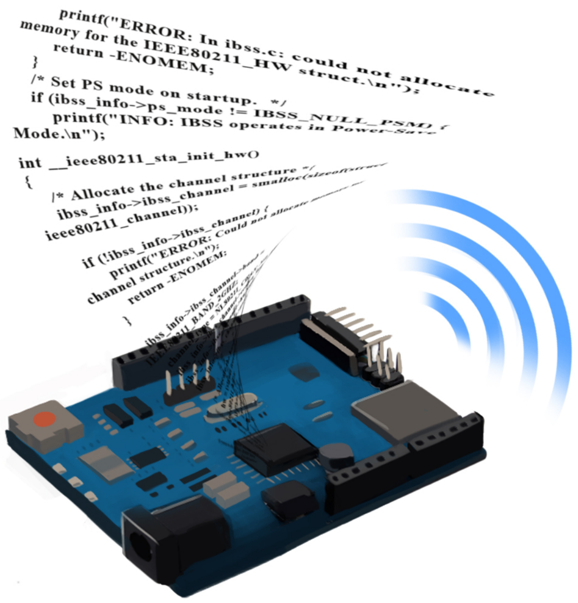
Ioannis Glaropoulos, Vladimir Vukadinovic, and Stefan Mangold

Abstract—We believe that the existing 802.11 standards can be optimized (especially in terms of energy-efficiency) to make Wi-Fi suitable for a wide range of IoT applications. However, there is a lack of low-cost embedded platforms that can be used for experimentation with 802.11 MAC protocol. The vast majority of low-power Wi-Fi modules for embedded systems has closed source firmware and protocol stack implementations, which prevents implementation and testing of new protocol features. Here we describe *Contiki80211*, an open source 802.11 radio link layer implementation for Contiki OS, whose purpose is to enable experimentation with 802.11 MAC layer management mechanisms on embedded devices, such as sensor motes and IoT smart objects. Contiki80211 implements a number of optimizations in order to run on hosts that are constrained in terms of memory and processing power. We evaluate its performance (memory usage, interrupt processing latency, etc.) on an embedded platform.

Keywords—IEEE 802.11, IoT, Contiki, prototyping, experimentation.

1 Introduction

The future Internet of Things (IoT) will provide global IP connectivity to a broad variety of devices, such as entertainment electronics, wearable sport gadgets, home appliances, and industrial sensors. Some of these devices are portable, battery-powered, and need to connect wirelessly to surrounding devices and Internet gateways. Different wireless standards are porposed as the solution for today's IoT. Zigbee, which is based on the IEEE 802.15.4 standard [1], is often referred to as a wireless technology of choice for home and building automation, smart metering, and IoT in general because of its simplicity and energy-efficiency. Z-Wave [2] is another technology that targets similar applications and environments with emphasis on home automation. Both



```
        printf("ERROR: In ibss.c: could not allocate
memory for the IEEE80211_HW struct.\n");
        return -ENOMEM;
    }
    /* Set PS mode on startup.  */
    if (ibss_info->ps_mode != IBSS_NULL_PSM) {
        printf("INFO: IBSS operates in Power-Save
Mode.\n");

int __ieee80211_sta_init_hw()
{
    /* Allocate the channel structure */
    ibss_info->ibss_channel = smalloc(sizeof(struct
    ieee80211_channel));

    if (!ibss_info->ibss_channel) {
        printf("ERROR: Could not allocate memory for
    channel structure.\n");
        return -ENOMEM;
    }
```

Fig. 1. Open-source Contiki80211 driver provides 802.11 connectivity to IoT motes (© Disney).

Zigbee and Z-Wave provide meshing capabilities, which are required by many IoT applications. Although it does not support meshing, Bluetooth Low Energy (BLE) is also a candidate technology for IoT. The comparative advantage of BLE, besides its low energy consumption, is its support in the majority of todays smartphone hardware and protocol stacks.

These technologies, however, do not cover the entire spectrum of IoT devices and applications. Wi-Fi, which is based on IEEE 802.11 standard [3], dominates the consumer electronics segment. IoT devices that need to connect to smartphones, tablets, TVs, set-top boxes, game consoles, and toys would benefit from Wi-Fi connectivity. Therefore, it is expected that future consumer electronics market will boost Wi-Fi's presence on the IoT market. The economy of scale and the possibility of reuse of the existing Wi-Fi infrastructure offer key cost savings and facilitate faster deployment with Wi-Fi than with competing technologies. Furthermore, Wi-Fi has the ad-

• *I. Glaropoulos is with the Access Linnaeus Centre, KTH, Royal Institute of Technology, Stockholm, Sweden, 10044. His work was done while at Disney Research. E-mail: ioannisg@kth.se.*

• *V. Vukadinovic and S. Mangold are with Disney Research Zurich. E-mail: {vvuk,stefan}@disneyresearch.com.*

This work is partially funded by European Community's Seventh Framework Program, area "Internet Connected Objects", under Grant No. 288879, CALIPSO Project – Connect All IP-based Smart Objects. The work reflects only the authors views; the European Community is not liable for any use that may be made of the information contained herein.

vantage of native compatibility with IP, which is the key enabler for IoT: IP eliminates the need for expensive gateway solutions to connect IoT devices to the Internet. Furthermore, some sensors that operate at high sampling rates, such as those used in seismic monitoring and imaging, may generate large amounts of data that cannot be transmitted using ZigBee and Z-Wave due to their limited transmission rate, but can easily be transmitted by Wi-Fi. The feasibility of connecting battery-powered sensors to the IoT using commercially available Wi-Fi chips has been demonstrated in [4]. In [5], the authors share their experiences of using off-shelf Wi-Fi modules to connect *things* to the Web of Things.

The energy consumption of Wi-Fi is relatively high compared to ZigBee, Z-Wave and BLE and it may quickly drain the battery of a device. For cost and convenience reasons, long battery recharge/replacement cycles are preferred not only for sensors but also for consumer electronics. There have been some notable improvements in radio hardware and many low-power Wi-Fi chips with energy-efficient radio transceivers have appeared in the market. The 802.11 MAC protocol, however, is inherently energy-hungry. One of the major sources of unnecessary energy consumption in 802.11 MAC is idle listening, which consumes energy even when there is no traffic exchange in the network. To alleviate the problem, the MAC layer management entity (MLME) of the 802.11 standard [3] includes a power-saving mode (PSM). PSM allows an idle 802.11 station to transit to a low-power *doze* state by switching off its radio transceiver. However, PSM has been designed for single-hop communication and, therefore, it is not suitable for IoT applications that assume multi-hop communication.

In [6], [7], we describe how 802.11 PSM can be optimized for multi-hop communication with minimal amendments to the standard. One problem that we faced while testing the proposed amendments was the lack of embedded platforms that can be used for experimentation with 802.11 MAC protocol. On one side, the vast majority of low-power Wi-Fi modules for embedded systems has closed-source firmware and protocol stack implementations, which prevents the implementation and the testing of new MAC protocol features. On the other side, popular embedded operating systems for IoT, such as Tiny OS [8] and Contiki OS [9], are developed for ZigBee-enabled motes and, therefore, they lack kernel libraries and drivers for Wi-Fi devices.

In this paper, we describe *Contiki80211*, an open source 802.11 radio link layer (MAC and 802.11 device driver) implementation for Contiki OS, one of the most popular operating systems for embedded systems and IoT. The purpose of Contiki80211 is to enable experimentation with 802.11 MAC layer management mechanisms on embedded platforms, such as sensor motes and IoT smart devices. The integration of Contiki80211 with the Contiki network protocol suite enables researchers to run and evaluate IETF protocols for IoT, such as RPL and CoAP, on top of an 802.11 radio link layer. Contiki80211 supports ad hoc (IBSS) mode for direct device-to-device communication and power saving mode (PSM) for radio duty-cycling. In order to provide maximum flexibility for experimentation with low-cost off-the-shelf hardware, Contiki80211 uses an Qualcomm Atheros AR9170-based radio for which an open-source firmware is available. Contiki80211 implements a number of optimizations in order to run an otherwise resource-hungry 802.11 MAC layer on hosts that are constrained in terms of memory and processing power.

The rest of the paper is organized as follows: Section 2 describes the software implementation of Contiki80211 and its integration into the networking protocol stack of Contiki OS. In Section 3, we evaluate the Contiki80211 code performance on an embedded platform. Section 4 gives a brief overview of related work, while Section 5 concludes the paper.

2 CONTIKI80211

We implemented Contiki80211 on a hardware platform shown in Fig. 2. The platform consists of an Arduino Due board (ARM Cortex-M3 MCU, 96 KB SRAM, 512 KB Flash) and an 802.11 interface attached to it via USB. The 802.11 interface module is based on the Atheros AR9170 chip. Contiki80211 provides link layer functionalities to the Contiki's micro IP stack (uIP6) and uses an API provided by the AR9170 firmware to exchange commands, asynchronous responses, and frames with the 802.11 interface on the AR9170 device (Fig. 2). The implementation of Contiki80211 was facilitated by the existence of *carl9170*, an open-source AR9170 driver for the Linux kernel [10]. carl9170 implements a number of features, such as the support for BSS (infrastructure) and IBSS (ad hoc) operation modes, rate-control algorithms, and MAC encryption. It, however, lacks a PSM implementation for the ad hoc mode. A depreciated version of carl9170 driver, called *Otus* driver, includes an incomplete non-functional implementation of PSM for the ad hoc mode [11]. Contiki80211 is partialy based on the carl9170 driver. Its implementation of PSM for ad hoc mode is an extension of the incomplete Otus driver code.

Contiki80211 consists of three functional blocks, as shown in Fig. 3:

- platform-independent IEEE80211Lib, which implements 802.11 IBSS management (scan, join, create, leave), IBSS parameter configuration, frame generation and parsing, and a power saving mode (PSM) scheduler
- AR9170 radio driver, which manages TX and RX queues, handles commands/hardware responses to/from the 802.11 interface, and implements the lower-level PSM functionality, namely the RF duty-cycling
- AR9170 USB driver, which implements routines for installation and enumeration of the 802.11 interface, and allocation of the end-points for communication with the interface over USB.

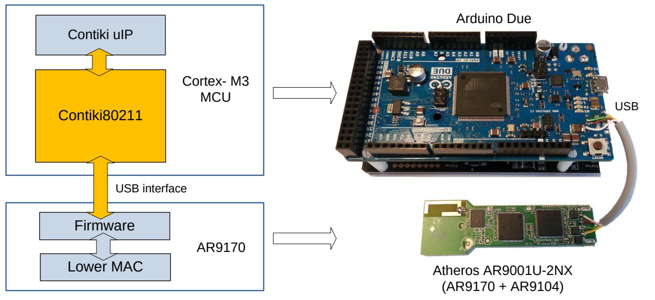


Fig. 2. Hardware platform on which we implemented Contiki80211.

Each of the three functional blocks is described in the following:

IEEE80211Lib: The IEEE80211Lib connects the uIP protocol stack with the radio link layer. It encapsulates IP packets into 802.11 frames, generates 802.11 management frames, and parses incoming 802.11 frames. It implements an 802.11 connection manager that is responsible for network configuration and management, including network scanning, joining, creating or leaving an IBSS network. Finally, IEEE80211Lib implements the PSM algorithm including the generation of ATIM frames and maintenance of a list of awake neighbors [6].

AR9170 Radio Driver: The AR9170 radio driver is responsible for dynamic MAC configuration through the AR9170 command API and for frame transmission and reception. The core of the AR9170 radio driver is the AR9170 scheduler. The scheduler is polled by the Contiki OS together with other processes using round-robin policy. At each execution round the scheduler inspects the contents of the command, TX and RX queues and dispatches the next-in-line task prioritizing command over TX/RX packet processing. The scheduler accesses the TX and RX queues in a round-robin fashion, so as to guarantee fairness in packet handling. The AR9170 radio driver implements the 802.11 IBSS PSM duty-cycling using a high-resolution Timer Counter (TC) interrupt routine that schedules state transitions in real-time. State transitions notify the PSM algorithm of the IEEE80211Lib to perform the required actions.

The AR9170 driver waits for a pre-TBTT interrupt from the 802.11 interface in order to prepare the 802.11 protocol for the upcoming ATIM window. The selection of the optimal pre-TBTT period duration takes into consideration the speed of the host's CPU, as well as the processing overhead of the routines to be executed in the beginning of each beacon interval. A long pre-TBTT period guarantees that all tasks will be executed in time, but it decreases the duration of the TX/RX Window in each beacon interval. We implemented a closed-loop control scheme that dynamically adjusts the pre-TBTT period duration based on the percentage of beacon intervals, in which the required tasks were executed in time.

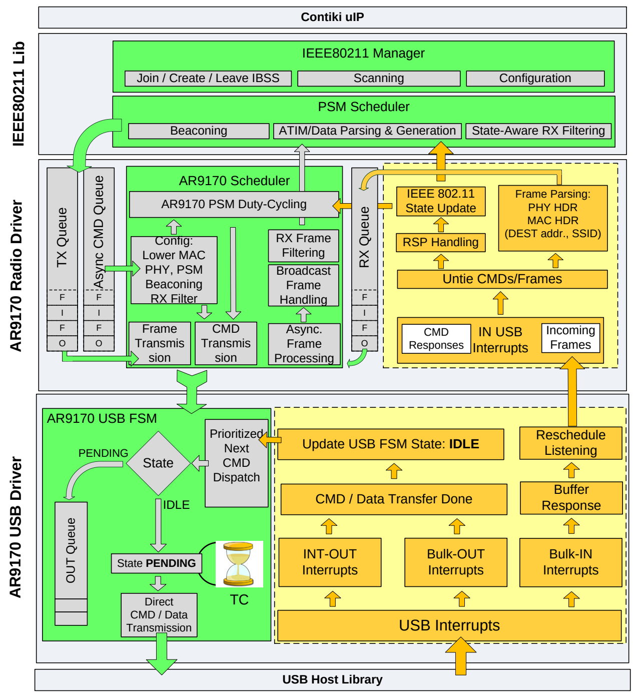


Fig. 3. Contiki80211 consists of three functional blocks: IEEE80211Lib, AR9170 radio driver, and AR9170 USB driver. Yellow boxes represent functions executed within interrupt context. Green boxes represent function executed asynchronously by the Contiki scheduling system.

AR9170 USB Driver: The AR917 radio driver requires an underlying USB driver to handle the communication between the host and the 802.11 interface. We implemented the USB driver using Atmel's USB host library [12], which provides platform-independent routines for the installation of a USB device, enumeration, and allocation of the required endpoints for communication with the device, as well as the low-level USB on-the-go driver for Arduino Due. On top of these routines, we implemented functions for reading and writing from/to the allocated endpoints.

The state machine of the USB driver is shown in Fig. 3. The driver is in the PENDING state, when the transfer of a USB data chunk pushed to the USB line is not yet completed. The driver transits to the IDLE state when the transfer is completed. A simple flow-control is implemented to prevents a data chunk to be sent before the previous transfer is completed. The outgoing data chunks may have to be buffered at the USB driver. Pending commands are given priority over outgoing packets. In order to accelerate outgoing USB transfers, a new transfer is dispatched directly inside the OUT completion callback function, within the interrupt context.

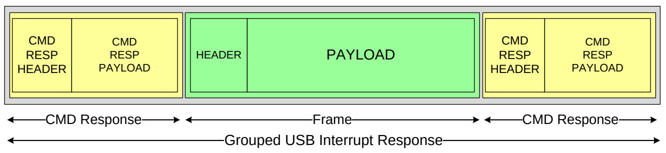


Fig. 4. A grouped USB interrupt response example containing two command responses and a data frame.

2.1 Resource Optimization

Both carl9170 and Otus have been designed for non-constrained hosts, such as PCs, while Contiki80211 is expected to run on resource-constrained hosts. In the following, we describe some of the most important resource optimizations that we made in Contiki80211.

2.1.1 Minimizing TX/RX Buffer Space

carl9170 allocates 16 TX-RX buffer pairs, where USB resource blocks (driver commands, command responses and incoming/outgoing frames) are stored before being processed. Each buffer has a length of 2 KB to accommodate the maximum possible MAC PDU length. This results in 64 KB of RAM being reserved for the buffers only, which is two thirds of the Cortex-M3 total SRAM memory. Instead, Contiki80211 uses a single TX-RX buffer (Fig. 3) to minimize the memory requirements. To handle heavy incoming traffic, it implements dynamic RX buffer space allocation, which increases the processing load in favor of low memory usage. In Section 3, we evaluate the impact of dynamic buffer allocation on processing latency for RX interrupts that are coming from the 802.11 interface.

2.1.2 Minimizing Interrupt Processing Load

A major challenge for constrained hosts is the CPU load of the code executed in the context of an interrupt coming from the 802.11 USB interface. If the interrupt execution time is long, there is a high risk of missing subsequent USB interrupts, which are, instead, handled as a single interrupt by the CPU. However, this particular *interrupt response*, i.e. the incoming data associated with this interrupt, will contain multiple interrupt responses from all these subsequent USB interrupts. In other words, there is a risk of a *grouped* interrupt response, containing multiple frames, or frames grouped together with command or hardware responses from the AR9170 device (Fig. 4). This incurs a significant effort to untie the different response units from each other. Hence, the interrupt context code must be kept as minimal as possible. Contiki80211 reduces the interrupt processing load by moving the processing of incoming data frames outside the interrupt context, except for the lightweight checks for PHY and MAC header errors. The frames are then buffered and processed off-line by the AR9170 scheduler. Command and hardware responses are still handled on-line, but the handling consists only of driver and 802.11 state updates (Fig. 3).

2.1.3 State-Aware RX Filtering

The described method of minimizing the interrupt processing load has a serious drawback in case of intensive data traffic. Since all frames must be buffered, the RX queue may overflow. In addition, moving a part of frame processing outside interrupt context provides no guarantee for the frame handling delay. ATIM frames, however, need to be handled fast, so that the PSM state machine is updated before the end of the ATIM window. These problems are solved by dynamically enforcing low-level frame filtering through the AR9170 command API, so that specific frames are filtered-out by the 802.11 interface, without reaching the host's MCU. RX filtering is PSM state-aware: Depending on the 802.11 PSM state, the AR9170 accepts or drops certain frame types:

- during ATIM window AR9170 accepts only 802.11 MGMT frames
- during TX/RX window AR9170 accepts only 802.11 DATA frames with the default SSID, or MGMT frames, with the exception of BCN and ATIM frames
- during Soft BCN window [7] AR9170 accepts both DATA and MGMT frames

This ensures that only relevant traffic interrupts the host's MCU.

2.1.4 Interrupt Response Parsing

The above described techniques for reducing the CPU load do not entirely eliminate the risk of grouped interrupts from the 802.11 interface. Therefore, an efficient interrupt response parsing algorithm is implemented, which uses the characteristic header patterns of USB command/hardware responses. The existence of a command response block inside the interrupt response is detected due to presence of the header. The command responses are extracted from the USB interrupt response, and the remaining blocks are treated as incoming frames. Since frame headers lack such characteristic pattern, consecutive frames are hard to delimit. This is done off-line by inspecting the payload of the received frames. Notice, that such parsing is not needed in the original carl9170 driver because interrupts are rarely missed or grouped when host's CPU is fast.

2.1.5 Prioritization of USB Interrupts

To minimize the risk of missed USB interrupts, we increase the priority of USB interrupts against interrupts from other peripherals, such as timers, system clocks, and serial ports. This, however, increases the risk of performance degradation for time-sensitive applications that depend on those peripherals.

2.1.6 USB Transfer Timeouts

Due to long interrupt processing delays on CPU-constrained hosts, it is possible that outgoing transfer completion interrupts are missed. This occurs when multiple interrupts of the same type (e.g. USB) overwrite each other. A missed transfer completion interrupt could result in a deadlock of the USB driver state machine. To avoid this, we implemented a timeout handler for each pending outgoing transfer: If the transfer completion interrupt is lost, the driver waits for a predefined period, after which the USB state machine transits to the IDLE state. On our Cortex-M3 platform, a dedicated high-precision timer counter is used to implement this policy.

2.1.7 32-bit Memory Copy

On the 32-bit Cortex-M3 MCU, 32-bit operations are carried out at system clock frequency. Since Contiki80211 functions often copy and move memory blocks between different locations in RAM, we accelerate the code performance by redefining `memmove` and `memcpy`, so they always execute on 32-bit integers. We have measured an almost four-fold decrease in processing delay compared to the standard `memmove` and `memcpy` functions, were copying is performed byte-by-byte: For a 1000 B array, memory copy is completed in $\sim 10\mu$s instead of $\sim 40\mu$s.

2.2 Integration with Contiki OS

Contiki OS is one of the most popular operating systems for embedded systems and IoT. Its protocol stack is optimized to provide wireless connectivity to resource-constrained devices. It fully supports IPv6, RPL routing protocol for low-power and lossy networks, and the Constrained Application Protocol (CoAP), which makes it well suited for the development of a wide range of IoT applications.

2.2.1 Contiki Porting

The porting of Contiki OS to the Arduino Due platform involves re-implementation of CPU-dependent Contiki code in order to run on the Cortex-M3 MCU. The most important of the ported libraries include the software for the basic peripherals, such as the system clock, UART, SPI and USART lines, RTimer, Watchdog and Flash controllers. An efficient high resolution RTimer library is essential for the Contiki80211 implementation. High resolution, however, decreases the time window during which events can be scheduled. To alleviate this problem we redesign the RTimer library using two timer counter (TC) channels on the Cortex-M3 board and the maximum time precision without any pre-scaling, allowing us to schedule interrupts with 12 ns resolution. The first TC is solely used to count the universal time, while the second is used to schedule short-term events. The overflow of the first TC increments the MSB part of the universal time, which is saved in a 64-bit integer; since the two counters are separate, the overflow does not affect the short-term event scheduling. In addition, we enhance the RTimer library to support parallel scheduling of multiple RTimer interrupts – non-supported feature in Contiki OS – which facilitates the implementation of the AR9170 scheduler and the AR9170 USB state machine.

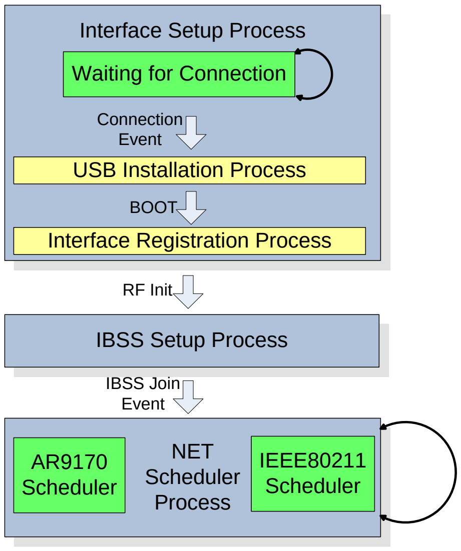


Fig. 5. Integration of AR9170 and 802.11 system processes into Contiki OS.

2.2.2 Process Integration

The integration of the Contiki80211-related processes with the Contiki OS process scheduling system is shown in Fig. 5. The `Interface Setup Process` waits for the 802.11 interface connection: a USB connection event triggers the process to invoke the USB installation procedure and firmware upload. The BOOT event invokes interface registration, MAC and RF initialization. Triggered by the *RFinit* signal, the `IBSS Setup Process` is invoked, which attempts to join the default IBSS network or to create it, depending on the network scanning result. A successful IBSS setup triggers the start of the `NET Scheduler Process`, where the aforementioned AR9170 scheduler and the PSM scheduler are integrated. All system processes exchange signaling messages using the default Contiki inter-process event-posting mechanism.

2.2.3 Code Integration

We integrated Contiki80211 into the Contiki's network protocol stack (`NETSTACK`). `NETSTACK` organizes the network modules into a complete protocol stack covering all traditional OSI layers.

On the top level of the `NETSTACK`, at the `NETSTACK_NETWORK` layer, we implemented `IEEE80211_net` (Fig. 6). `IEEE80211_net` connects the underlying Contiki80211 modules with the Contiki's uIP stack. Unlike `6LoWPAN`, which is used with the

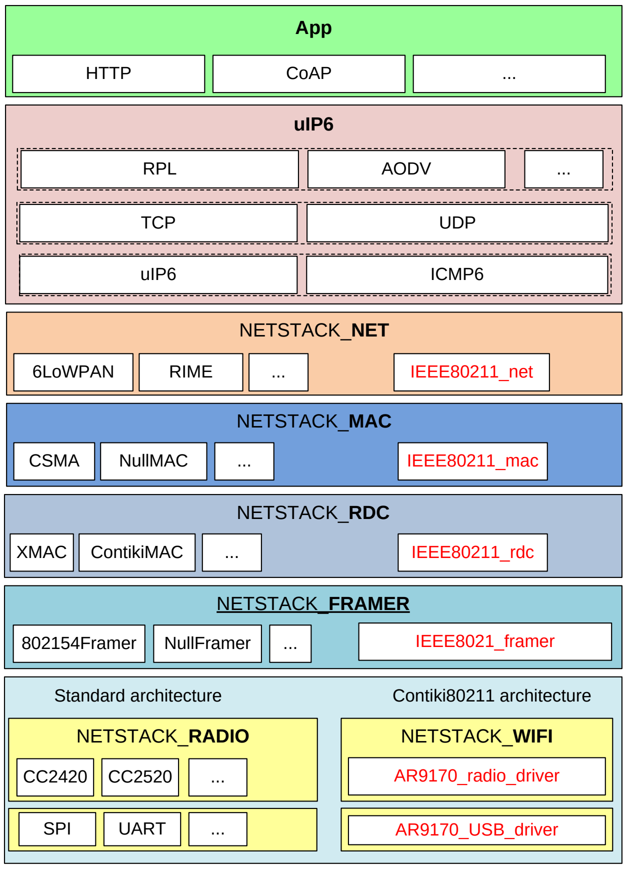


Fig. 6. The standard and the 802.11-enabled Contiki network stacks.

802.15.4 radio link layer, `IEEE80211_net` provides neither IPv6 packet compression nor fragmentation support since 802.11 frames can accommodate long IPv6 packets. On the `NETSTACK_MAC` layer, we introduce the `IEEE80211_mac` module that implements the IEEE80211Lib functionality. `IEEE80211_mac` delivers IP packets (**MAC frame payloads**) to the underlying `IEEE80211_rdc` module, which uses `IEEE80211_framer` to encapsulate them into 802.11 frames. At the opposite direction, `IEEE80211_rdc` performs MAC address filtering and duplicate frame detection and delivers IP packets to the `IEEE80211_mac`. Below the `NETSTACK_FRAMER` layer, we introduced the `NETSTACK_WIFI` layer that contains the `AR9170_radio_driver` and `AR9170_USB_driver` modules.

Note that although the higher uIP and application layers are supposed to be oblivious to the radio link layer modules, they are by default parametrized for 802.15.4 and, therefore, must be modified to meet the requirements of the 802.11 radio link layer. This involves an expansion of the Contiki packet buffer structures (`uIPbuf`, `packetbuf`), an increase of the maximum UDP and IPv6 packet sizes, and the adoption of an Ethernet-style link-layer address (`uip_lladdr_t`) structure.

3 Performance Evaluation

We evaluate the effect of the various optimizations described in Section 2.1 on the performance of Contiki80211. We consider a scenario where a Contiki80211-enabled mote shown in Fig. 2 is placed inside a building of an university campus. The mote does not transmit any 802.11 frames, but it receives and processes all incoming 802.11 traffic, which is heavy since the experiments were performed during busy hours. In each experiment, we measure the Contiki80211 performance (memory usage, interrupt processing latency, etc.) on an ARM Cortex-M3 MCU during a time interval of 120 minutes.

3.1 Performance of RX Buffer Space Allocation

We investigate the effect of the dynamic buffer allocation for the incoming USB interrupt responses, which is described in Section 2.1.1. Fig. 7 shows the memory usage and RX interrupt processing delay for three buffer allocation policies: the *static* policy, where 16 buffers (2 KB each) are reserved at compilation time using the Memory Block Allocator (`memb`) of Contiki OS, the *dynamic* policy, where buffers are allocated on-demand in real time on the Managed Memory (`mmem`) of Contiki, and the *hybrid* policy, where a buffer of maximum length is allocated at the time of re-scheduling listening at the bulk-in USB endpoint. As expected, the static policy results in the shortest processing delay because it does not require to reserve memory for command and/or frames responses inside interrupts. In addition, it does not require copying the interrupt response content to the Contiki buffers before proessing, as the static memory blocks are thread-safe and can serve as on-demand Contiki buffers. The performance improvement of the static policy is achieved, however, at the expense of high memory usage. The dynamic policy minimizes memory usage, but increases the processing delays as memory must be allocated and the interrupt response content bust be copied inside interrupt context. The hybrid policy decreases the interrupt process delay as it schedules the resource allocation to be executed once the interrupt routine is terminated. In the event of a subsequent interrupt arrival before the resource allocation, the hybrid policy reduces to the dynamic one, however, since the frequency of such events is low, the average performance of the hybrid policy is higher, in the expense of slightly higher memory demand, as the a-priory resource allocation account for the maximum possible size of the interrupt response.

3.2 Performance of Interrupt Response Parsing

The processing of interrupt responses from the 802.11 interface consists of three stages: *usb handling*, i.e. resource

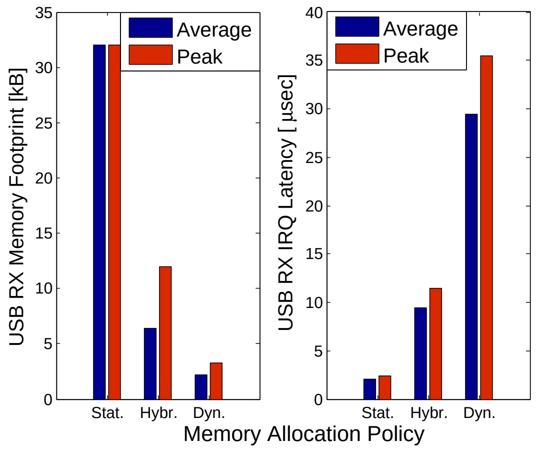


Fig. 7. Memory usage and interrupt processing delay for various buffer allocation policies.

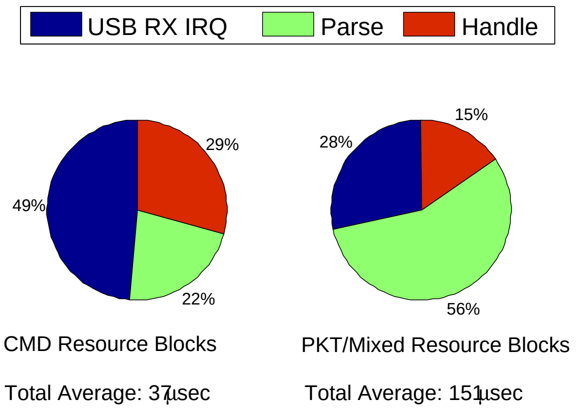


Fig. 8. Processing costs for various stages of interrupt response processing.

allocation, copy, and re-scheduling listener, *parsing* of the interrupt response, and *handling* of command/hardware responses and/or frames inside the interrupt. Fig. 8 shows the CPU processing cost associated with each of the stages. Interrupt responses that contain only command/hardware responses are short and the command response header pattern is immediately detected, which leads to low parsing delays. Interrupt responses that contain frames require longer parsing time. This is because the complexity of pattern search increases linearly with the length of the interrupt response. Clearly, in such interrupt responses, parsing dominates processing delay. When 802.11 interface is in fully-operational mode, most of the interrupt responses contain received frames and, therefore, the dynamic buffer allocation policy is optimal because it decreases the memory usage without causing a significant processing delay.

The results in Table 1 show the benefits of the interrupt response parsing described in Section 2.1.4 by

Parsing Algorithm	Parsing Delay	Frequency of Lost Commands
`carl9170`	14 μs	5.43%
`contiki80211`	100 μs	0.02%

TABLE 1
Parsing efficiency for the original and the optimized interrupt response parser.

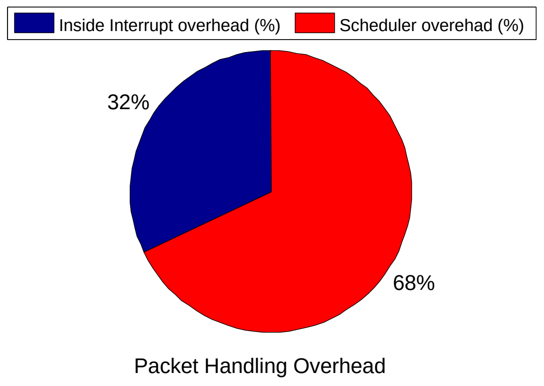


Fig. 9. Online and offline frame processing overhead.

comparing it with the original `carl9170` parser, which simply examines the beginning of an incoming interrupt response for a command response header pattern and, consequently, may fail to detect the presence of command responses inside a grouped interrupt response. Although such simple parser minimizes the interrupt processing delay, it results in a much higher frequency of lost command responses. This occurs due of the long usb handling and parsing/handling times for large frames inside the interrupt context, which increase the probability that the next incoming command and/or frame response(s) are grouped. The results show that the advanced interrupt response parser significantly reduces the frequency of lost command/hardware responses.

Fig. 9 shows CPU processing overhead for online and offline frame handling operations, as illustrated in the AR9170 radio driver diagram (Fig. 3). Since the offline handling constitutes the larger part of the total frame handling overhead, the decision to do the remaining frame handling offline results in a significant decrease in the interrupt response processing delay.

3.3 Performance of State-Aware RX Filtering

Contiki80211 employs the state-aware RX filtering described in Section 2.1.3. Here we evaluate its performance. As shown in Table 2, in the absence of the filtering mechanism the probability of RX buffer overflow is non-negligible. In the university campus scenario with a heavily background traffic, Contiki80211 needs to store and process all decoded management and data frames (including retransmissions) regardless of their origin. Buffer overflows occur at traffic bursts when the driver is

	Packet Loss (overflow)	Lost Command Responses	Late ATIMs [ATIM window: 10 ms, 20 ms, 40 ms]
RX filtering	∼0%	0.02%	0.05%, 0.01%, ∼0%
no filtering	1.60%	0.04%	5.54%, 0.91%, 0.01%

TABLE 2
Parsing efficiency for the original and Contiki80211 interrupt response parser.

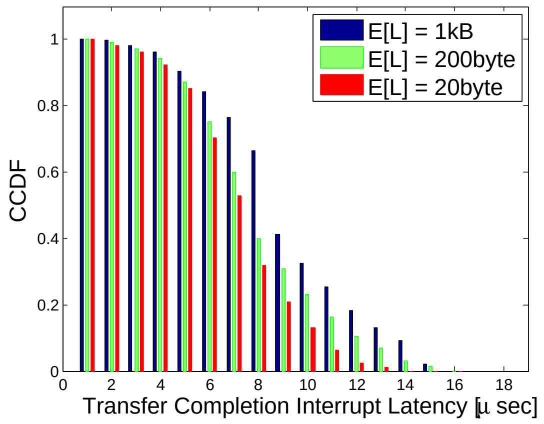


Fig. 10. Complementary CDF of USB transfer completion interrupt latencies for different expected transfer lengths.

unable to process all incoming traffic in time. In addition, in case of long incoming frames, the probability of lost command responses slightly increases.

We conducted an experiment similar to the one in [7] to measure the probability that an ATIM frame is not processed before the expiration of the current ATIM window. This probability depends on the length of the ATIM window. As shown in Table 2, the filtering of DATA frames before they reach the host prevents the RX queue of the AR9170 radio driver from growing large and, therefore, increases the probability for ATIM frames to be processed in time.

3.4 USB Transfer Timeouts

As discussed in Section 2.1.6, USB driver deadlocks are avoided with the help of a timeout mechanism that forces the USB driver to transit from the pending to the idle state. We aim at optimizing the required timeout duration in order to minimize the time wasted if a transfer completion interrupt is lost. The time required for the host to receive the transfer completion interrupt from the 802.11 interface depends on the USB host implementation on the Cortex-M3, on the USB protocol implementation on the AR9170 chip, on the prioritization of USB interrupts, and on the CPU load at both the host and the 802.11 interface. Fig. 10 shows the empirical distribution of the transfer completion interrupt latency, as computed in our measurements. This latency depends on the size of the outgoing USB transfer L. Based on Fig. 10, we set the timeout threshold to 16 μs, which tightly upperbounds the latencies. Besides avoiding long deadlocks, such a tight timeout threshold enables fast re-attempts of USB outgoing command transfers, whose completion interrupts get lost. The probability of such losses was measured to be in the order of $\sim 10^{-3}$, showing that the re-transmission overhead is kept low.

3.5 Prioritization of USB Interrupts

As discussed in Section 2.1.5, increasing the priority of USB line interrupts over the other interrupt sources in our platform – namely the system *tick* and timer counters, the digital input PINs, the watchdog interrupt, the UART, SPI and TWI interface, and the random generator – can improve the performance of Contiki80211. We modified the USB interrupt priority level by setting the `UHD_USB_INT_LEVEL` flag to 1, which results in a priority level that is second only to the timer counters. Table 3 shows the effect of the increased USB interrupt priority level on the rate of USB completion timeout of outgoing transfers, and on the percentage of grouped incoming interrupt responses. The results indicate that, the performance increase is significant. However, since the USB interrupt delays are relatively high, as we show in Fig. 9, one must guarantee that the other interrupting sources in the MCU do not need to perform operations with stricter time constraints, compared to the USB.

4 Related Work

Here we give an overview of open-source Linux-based platforms that provide certain flexibility to modify or implement new 802.11 MAC/MLME algorithms, as well as of (typically closed-source) 802.11 modules and development boards for embedded systems. Our focus is on low-cost embedded hardware/software rather than on expensive FPGA boards, such as WARP [13], which provide full flexibility to develop new wireless protocols, but no possibility to test them in an embedded environment.

We have surveyed a number of open-source Linux drivers for Wi-Fi chips of various vendors, including Realtek, Broadcom, Intel, and Atheros, in order to asses their MAC and MLME implementations — PSM in particular because power saving is essential for energy-constrained sensor nodes and smart objects. Although Linux drivers are not written for such devices, they may be a good starting point for developing a more optimized code. In some drivers, such as in the *brcm80211* [14] for Broadcom chips, the PSM-related code is limited to the

	USB-out completion timeout	Percentage of grouped responses
`UHD_USB_INT_LEVEL=1` highest	∼0%	0.22%
`UHD_USB_INT_LEVEL=5` lowest (usual)	0.28%	4.78%

TABLE 3
The effect of prioritizing USB interrupts on the Cortex-M3 MCU.

configuration API, since the PSM is implemented on-chip. Most drivers, such as *iwlwifi* for Intel chips [15], *rtlwifi* for Realtek chips [16], *b43* for Broadcom chips [17], and *ath5k* [18], *ath9k* [19], and *carl9170* [10] for Atheros chips, do contain PSM implementations, but only for BSS (infrastructure) mode. In B43 and carl9170, the PSM is listed as a non-working feature, while in iwlwifi and rtlwifi, the PSM is claimed to work properly for BSS. A fully functional implementation of the 802.11 PSM for IBSS (ad hoc) mode is not present in any of the drivers. A driver that provides an incomplete PSM implementation for IBSS mode is *otus* [11]. Therefore, we used it as a basis for our code. The otus driver supports the AR9170 Atheros chip and it is currently in the staging state after being replaced by the carl9170 driver as of Linux kernel version 2.6.

There are several projects that have used the above mentioned open-source drivers to build experimental platforms for academic research. *OpenFWWF* [20] is envisioned as a platform for developing, prototyping and testing new MAC implementations. The project has developed a firmware code that runs on a Broadcom chip/NIC and provides an extended command and routines set for developing new MAC protocols. The firmware however, relies on the upper-MAC implementation of the b43 driver, which lacks PSM for IBSS mode. Similarly, the *Wireless Mac Processor* [21] is an application environment for developing and testing wireless MAC protocols. It provides a graphical tool for designing protocols' finite state machines, which are then used to build the corresponding device firmware codes. The tool requires BCM4311 or BCM4318 Broadcom chip/NIC and b43 driver.

Both OpenFWWF and WMP run on Linux PC platforms and are not best suited for experimenting with MAC protocols for constrained sensor nodes and smart objects. More promising development platforms for such devices and applications are 802.11-enabled MCU boards that are running OpenWrt [22], an open-source embedded operating system based on the Linux kernel. For example, *Arduino Yun* [23] and *Carambola 2* [24] are two OpenWrt-based modules that use Atheros AR9331 chip driven by the *ath9k* Linux driver. Our platform has two main advantages over [23] and [24]: it uses the AR9170 chip, which comes with the open-source firmware, and it is integrated into the Contiki OS and its state-of-art IoT protocol stack.

There is a number of low-power 802.11 modules for embedded systems on the market, such as Roving RN-131C/Wi-Fly, Gainspan GS2100, Texas Instruments CC3000, Atheros AR4100, Broadcom BCM4390, as well as development boards for IoT applications that integrate these modules, such as Flyport [25], SparkCore [26], WiSmart [27], and WICED [28]. These modules and development boards, however, are not suitable for experimentation with 802.11 MAC protocols and MAC layer management entity because they come with closed firmware and proprietary protocol stack implementations.

5 Conclusions

References

[1] *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Std., Rev. IEEE Std 802.15.4, 2006.

[2] "Z-Wave Alliance," http://www.z-wavealliance.org.

[3] *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std., Rev. IEEE Std 802.11-2012, 2012.

[4] S. Tozlu, M. Senel, W. Mao, and A. Keshavarzian, "Wi-fi enabled sensors for internet of things: A practical approach," *Communications Magazine, IEEE*, vol. 50, no. 6, pp. 134–143, June 2012.

[5] B. Ostermaier, M. Kovatsch, and S. Santini, "Connecting things to the web using programmable low-power wifi modules," in *Proceedings of the Second International Workshop on Web of Things*, ser. WoT '11, 2011, pp. 2:1–2:6.

[6] I. Glaropoulos, S. Mangold, and V. Vukadinovic, "Enhanced IEEE 802.11 Power Saving for Multi-Hop Toy-to-Toy Communication," in *IEEE Internet of Things (iThings)*, Beijing, China, 2013.

[7] V. Vukadinovic, I. Glaropoulos, and S. Mangold, "Enhanced Power Saving Mode for Low-Latency Communication in Multi-Hop 802.11 Networks," *Elsevier Ad Hoc Networks*, revised, available upon request.

[8] P. Levis, S. Madden, J. Polastre, S. R., K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for wireless sensor networks," in *Ambient Intelligence*, J. R. W. Weber and J. E. Aarts, Eds. Springer, Berlin, 2005.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proc. IEEE Int. Conf. Local Computer Networks*, Tampa, USA, 2004.

[10] "carl9170 - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/carl9170/. [Nov-2013].

[11] "otus - linux wireless," http://linuxwireless.org/en/users/Drivers/otus. [Nov-2013].

[12] "Atmel usb host library," http://www.atmel.com/Images/doc8486.pdf.

[13] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: A Flexible Wireless Open-Access Research Platform," in *Proc. EUSIPCO*, Florence, Italy, 2006.

[14] "brcm80211 - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/brcm80211/. [Nov-2013].

[15] "iwlwifi - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/iwlwifi/. [Nov-2013].

[16] "rtlwifi - Linux Wireless: realtek 802.11 drivers," http://http://wireless.kernel.org/en/users/Drivers/rtl819x/. [Apr-2014].

[17] "b43 - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/b43/. [Nov-2013].

[18] "ath5k - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/ath5k/. [Nov-2013].

[19] "ath9k - Linux Wireless," http://wireless.kernel.org/en/users/Drivers/ath9k/. [Nov-2013].

[20] "Open FirmWare for WiFi networks," http://www.ing.unibs.it/ openfwwf/. [Nov-2013].

[21] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware," in *Proc. IEEE INFOCOM*, Orlando, USA, 2012.

[22] "OpenWrt," https://openwrt.org/. [Nov-2013].

[23] "Arduino Yun," http://arduino.cc/en/Main/ArduinoBoardYun/. [Nov-2013].

[24] "Carambola 2," http://8devices.com/carambola-2/. [Nov-2013].

[25] "openPicus Flyport," http://www.openpicus.com/site/products/. [Nov-2013].

[26] "SparkCore," https://www.spark.io/. [Nov-2013].

[27] "WiSmart," http://www.econais.com/products/ec32lxx-family/. [Nov-2013].

[28] "WICED: Wireless Internet Connectivity for Embedded Devices," http://http://www.broadcom.com/products/wiced/wifi/. [Apr-2014].