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 MAC layer can be optimized (especially for energy-efficiency) to make Wi-Fi suitable for a wide range of IoT applications. However, there is a lack of low-cost embedded platforms to be used for experimentation with 802.11 MAC. The 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, optimized for resource constrained embedded platforms, whose purpose is to enable experimentation with 802.11 MAC layer management mechanisms on embedded devices.**

***Index Terms*—IEEE 802.11, IoT, Contiki OS, prototyping.**

I. 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. Several wireless standards are porposed for IoT including Zigbee, which is based on the IEEE 802.15.4 standard [1], Z-Wave [2] or Bluetooth Low Energy (BLE), targeting, respectively, smart metering applications, home and building automation, and connectivity with modern smartphone hardware. IoT devices that need to connect to smartphones, tablets, TVs, set-top boxes, game consoles, and toys would, however, benefit from IEEE 802.11 connectivity, as Wi-Fi technology dominates the consumer electronics segment. 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 advantage 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.

Wi-Fi energy consumption is, however, relatively high compared to ZigBee, Z-Wave and BLE and may quickly drain the battery of a device. There have been some notable improvements in radio hardware and many low-power Wi-Fi chips with energy-efficient radio transceivers have emerged. The 802.11 MAC protocol, however, is inherently energy-hungry, as a result of idle listening, which consumes energy even in the absense of traffic exchange. To alleviate the problem, the MAC layer management entity (MLME) of the 802.11 standard [3] includes a power-saving mode (PSM). In [4] we described how 802.11 PSM – originally designed for single-hom communication – can be optimized for multi-hop networking with minimal amendments to the standard. While testing the proposed amendments we were faced with a 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 [5] and Contiki OS [6], are developed for ZigBee-enabled motes and, therefore, they lack kernel libraries and drivers for Wi-Fi devices.

This paper describes *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, discussed in Section II, and evaluated in Section III, in order to run an otherwise resource-hungry 802.11 MAC layer on hosts that are constrained in terms of memory and processing power.

II. Contiki80211

We implemented Contiki80211, as part of the Contiki operating system [6], on a hardware platform (Fig. 1) consisting 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

I. Glaropoulos is with the Access Linnaeus Centre, Stockholm, Sweden. 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.

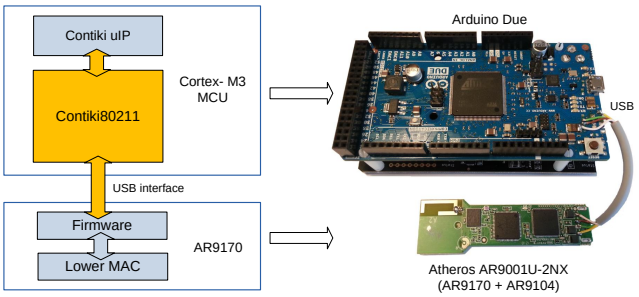

Fig. 1. Hardware platform for Contiki80211 implementation and evaluation.

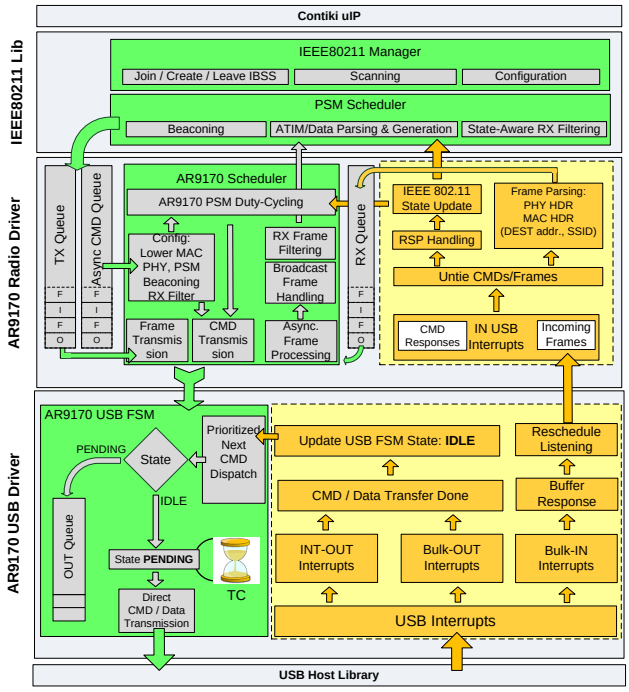

Fig. 2. Contiki80211 functional blocks. Yellow boxes represent functions executed within interrupt context. Green boxes represent function executed asynchronously by the Contiki scheduling system.

on the AR9170 device. Contiki80211 consists of three functional blocks (Fig. 2):

- 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 the TX and RX frame 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 handles the communication between the host and the 802.11 interface. It implements routines for installation and enumeration of the 802.11 interface, and allocation of the end-points for communication with the interface over USB.

A. Resource Optimization

The implementation of AR9170 radio and USB drivers is based on the open-source *carl9170* and *Otus* AR9170 Linux drivers [7] [8]. Both drivers, however, have been designed for non-constrained hosts, such as PCs, while Contiki80211 is expected to run on resource-constrained systems. In the following, we describe some of the most important resource optimizations that we made in Contiki80211.

1) Minimizing TX/RX Buffer Space: carl9170 allocates 16 TX-RX buffer pairs, where USB *resource blocks* (driver commands, hardware 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, resulting in 64 KB of memory being reserved for the buffers only, which would consume two thirds of the MCU's total RAM. Instead, Contiki80211 uses a single TX-RX buffer (Fig. 2). 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 III, we evaluate the impact of dynamic buffer allocation on processing latency for RX interrupts coming from the 802.11 interface.

2) Streamlined Interrupt Processing: 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 risk of missing subsequent USB interrupts, which are instead handled as a single interrupt. The incoming data associated with this *grouped* interrupt, will contain multiple interrupt responses from all these subsequent USB interrupts (e.g., multiple frames, or frames grouped with commands or hardware responses from the 802.11 interface), which are difficult to untie from each other. Therefore, 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. 2). Streamlined interrupt processing, however, does not entirely eliminate the risk of grouped interrupts. Therefore, the AR9170 radio driver implements an efficient interrupt response parsing algorithm, which uses the characteristic header patterns of USB command/hardware responses, in order to detect and extract them from the grouped interrupt response[1].

3) State-Aware RX Filtering: Off-line frame processing can, however, result in RX queue overflows in case of intense data traffic. Also, 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. We address this by dynamically enforcing low-level frame filtering,so that non-relevant frames are filtered-out by the 802.11 interface, without reaching the host MCU. RX

[1]Frame responses do not include such patterns and possible consecutive frames are untied by inspecting their payload.

filtering is PSM state-aware, as follows:

- 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 [4] AR9170 accepts both DATA and MGMT frames.

B. Integration with Contiki OS

Contiki network 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. 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 `NETSTACK`, at the `NETSTACK_NETWORK` layer, we introduce `IEEE80211_net` (Fig. 3), connecting the underlying Contiki80211 modules with the Contiki's uIPv6 stack. Unlike `6LoWPAN`, which is used with the 802.15.4 link layer, `IEEE80211_net` provides neither IPv6 packet compression nor fragmentation support, as 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 to the underlying `IEEE80211_rdc` module that 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`. On the bottom level the `NETSTACK_WIFI` layer includes the `AR9170_radio_driver` and `AR9170_USB_driver` modules.

III. Performace Evaluation

We evaluate the effect of the various optimizations described in Section II-A on the performance of Contiki80211. We consider a scenario where a Contiki80211-enabled mote shown in Fig. 1 is placed inside a building of an university campus. The mote receives and processes all incoming 802.11 traffic, which is heavy as the experiments were performed during busy hours. In each experiment, we measure the Contiki80211 performance (memory usage, interrupt processing latency, etc.) on the platform MCU during a time interval of 120 minutes.

A. Performance of RX Buffer Space Allocation

We investigate the effect of the dynamic buffer allocation for the incoming USB interrupt responses, described in Section II-A1. Fig. 4 shows the memory usage and RX interrupt processing delay for three allocation policies: the *static* policy, where 16 buffers (2 KB each) are reserved at compilation time, the *dynamic* policy, where buffers are allocated on-demand in real time, and a *hybrid* policy, where a maximum-sized buffer is reserved at the time of re-scheduling listening at the bulk-in USB endpoint. The static policy results in the shortest processing delay – at the expense of highest memory

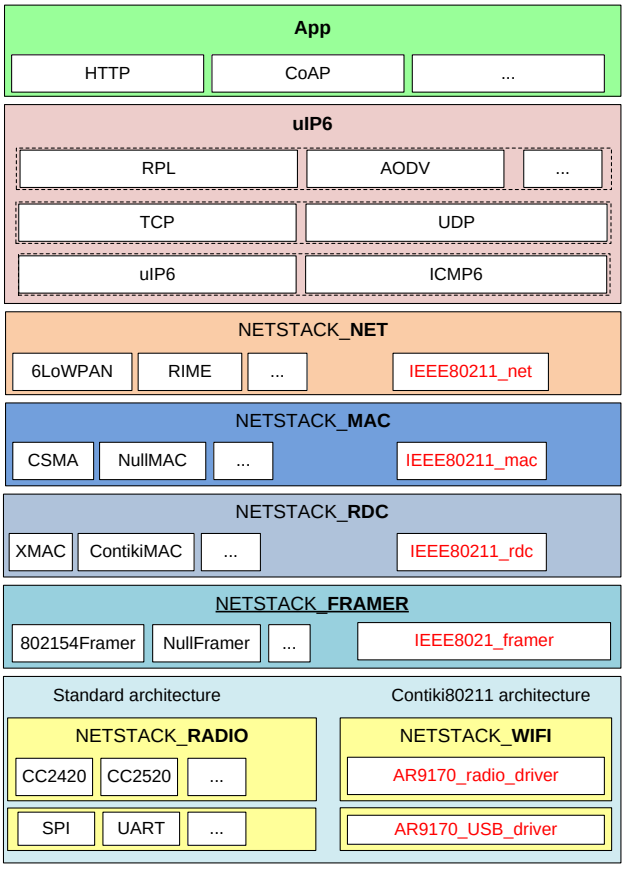

Fig. 3. Standard and 802.11-enabled Contiki network stacks.

usage – as it does not perform memory operations inside interrupt context[2]. The dynamic policy minimizes memory usage, but increases the processing delays as all memory operations are performed within 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. If a subsequent interrupt arrivs before the resource allocation, the ybrid policy reduces to the dynamic policy. However, since such events are rare, the hybrid policy performs better at the expense of slightly higher memory demand since the a-priory resource allocation account for the maximum possible size of the interrupt response.

B. Performance of Streamlined Interrupt

The processing of interrupt responses from the 802.11 interface consists of three stages: *usb handling*, i.e. memory allocation, copy, and re-scheduling listener, *parsing* to check for grouped responses, and *handling* of command/hardware responses and/or frames inside interrupt context Interrupt responses containing only command/hardware responses are short and the command response header pattern is immediately detected, leading to low parsing delays (Fig. 5). Interrupt responses containing frames require longer parsing time, because

[2]It does not require copying the interrupt response content to the Contiki buffers before prosessing, as the static memory blocks are thread-safe and can serve as on-demand Contiki buffers.

	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 I
EVALUATION OF THE PSM STATE-AWARE FRAME FILTERING.

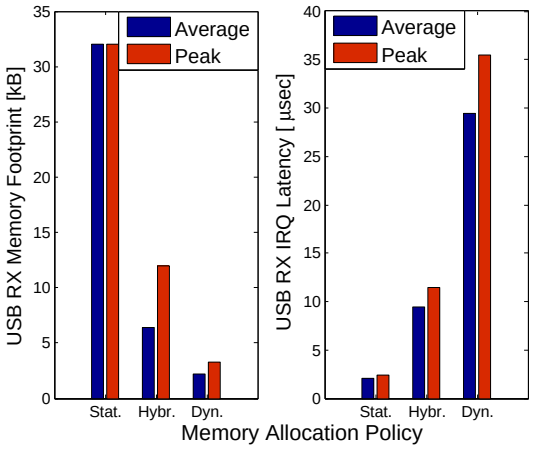

Fig. 4. Comparison of the different buffer allocation policies.

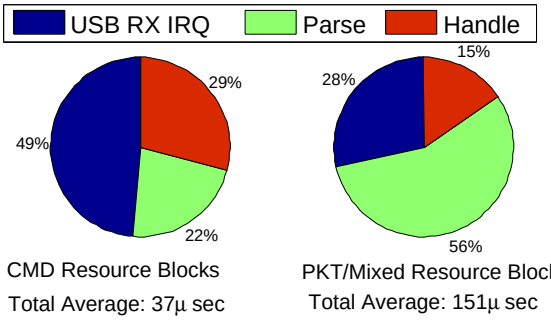

Fig. 5. Processing costs for various stages of interrupt response processing.

the complexity of pattern search is proporional to the length of the response, therefore, parsing dominates processing delay.

Table II shows that the interrupt response parsing described in Section II-A2 is more robust compared to the original `carl9170` parser, which, although minimizing the interrupt processing delay, as it simply examines the beginning of an incoming interrupt response for a command response header pattern, it may fail to detect the presence of command responses inside a grouped interrupt response. Table. III shows CPU processing overhead for online and offline frame handling operations, as illustrated in the AR9170 radio driver diagram (Fig. 2). As 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.

C. Performance of State-Aware RX Filtering

In Table I we evaluate the performance of the state-aware RX filtering described in Section II-A3. In the absence of the filtering mechanism the probability of RX buffer overflow is non-negligible. Contiki80211 stores and processes all decoded management and data frames (including retransmissions) regardless of their origin. Buffer overflows and response losses will, then, occur at traffic bursts when the driver is unable to process all incoming traffic in time.We conducted an experiment similar to the one in [4] 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 I, 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.

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

TABLE II
PARSING EFFICIENCY FOR THE ORIGINAL AND THE OPTIMIZED INTERRUPT RESPONSE PARSER.

	Processing Overhead (%)
Inside IRQ	32%
Outside IRQ	68%

TABLE III
ONLINE AND OFFLINE FRAME PROCESSING OVERHEAD.

IV. CONCLUSIONS

In this paper we presented Contiki80211, an open-source 802.11 radio link layer for the Contiki OS. Contiki80211 is fully integrated into the network stack of Contiki OS, enabling the experimentation and performance evaluation of IoT network stacks over 802.11. We described several resource optimizations that allow Contiki80211 to run efficiently on embedded devices with limited memory and processing power. In particular, a hybrid allocation policy for the RX queues results in a good trade-off between the memory requirement and interrupt processing delays. A careful assignement of driver operations to the interrupt and thread execution contexts is shown to guarantee a robust behavior of the driver. Finally, we showed that state-aware filtering of incoming packets increases the stability of Contiki80211.

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] 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*, 2014.

[5] 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.

[6] 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.

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

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