This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT

Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch

***Abstract*—As the Internet of Things (IoT) emerges, compact operating systems are required on low-end devices to ease development and portability of IoT applications. RIOT is a prominent free and open source operating system in this space. In this paper, we provide the first comprehensive overview of RIOT. We cover the key components of interest to potential developers and users: the kernel, hardware abstraction, and software modularity, both conceptually and in practice for various example configurations. We explain operational aspects like system boot-up, timers, power management, and the use of networking. Finally, the relevant APIs as exposed by the operating system are discussed along with the larger ecosystem around RIOT, including development and open source community aspects.**

***Index Terms*—Internet of Things (IoT), operating system, constrained networking, real-time system**

I. Introduction

The Internet is expanding with the advent of the Internet of Things (IoT)—billions of physical entities on our planet (and beyond) are expected to be instrumented and interconnected by open protocol standards. In particular, the IoT will harness next-generation sensors and actuators to interoperate with the physical world. Such cyber-physical systems will not only perform data acquisition and processing, but are also likely to control more and more elements of our environment. IoT devices will not only interconnect, but also extend communication beyond gateways, into today's Internet (e.g., the cloud) which has not dealt before with so many devices of marginal intelligence.

IoT devices in this context are very constrained [25] in terms of hardware resources. In particular, low-end IoT devices do not have enough resources to run conventional operating systems like Linux, BSD, or Windows, or even run optimized operating system (OS) derivatives like OpenWRT, uClinux, Brillo, or Windows 10 IoT Core. As resource limitations on low-end IoT devices are expected to last [37], [39], a variety of more compact operating systems were recently designed [29]. One of the prominent operating systems in this space is RIOT [10], which was written from scratch for low-end IoT devices. RIOT runs on minimal memory in the order of ≈10kByte, and can run on devices with neither MMU (memory management unit) nor MPU (memory protection unit). The goal of this paper is to provide an overview of RIOT, both from the operating system point of view, and from an open source software and ecosystem point of view.

Prior work [28], [29] has surveyed the space of operating systems for low-end IoT devices. Open source operating systems other than RIOT which target a similar range of IoT devices include Contiki [27], Zephyr [18], mbed OS [2], FreeRTOS [24], or TinyOS [36]. While it is not the goal of this paper to make a detailed comparison between RIOT and other operating systems for low-end IoT devices, we want to emphasize the following key distinctive aspects.

First, operating systems divide according to resource requirements. RIOT requires less memory and adapts to a wider range of architectures (8 to 32 bits) compared to most other OSes. Second, a characteristic feature amongst OSes in this domain relates to cross-platform hardware support. Some operating systems are tied to a hardware vendor and only target at a single hardware architecture. Finally, a distinction relates to what software features are provided. Some implementations offer a limited subset of OS components, e.g. FreeRTOS [24] is a scheduler; Arduino [1] is a hardware abstraction layer. On the contrary, as we describe in the following, RIOT provides the full set of features expected from an OS, ranging from hardware abstraction, kernel capabilities, system libraries, to tooling.

Up until now, RIOT has only been briefly described in [23]. In contrast, the contribution of this paper is a comprehensive overview and rationale of its building blocks, its interfaces to related IoT software, and its open source ecosystem.

First, we survey the main requirements for an OS running on low-end IoT devices (in § II). Next, we dive into the technical guts of the main areas and concepts of RIOT: the kernel, hardware abstraction, system initialization, power management, the timer subsystem, the networking subsystem, the main APIs and external libraries integration (in § III– XII). Finally, we describe the larger ecosystem around RIOT: the suite of tools for RIOT development, example OS configurations, code quality management processes, and open source community aspects (in § XIII– XVI).

E. Baccelli is with INRIA, France (e-mail: emmanuel.baccelli@inria.fr).

C. Gündoğan, P. Kietzmann, and T.C. Schmidt are with Hamburg University of Applied Sciences, Germany (e-mail: {cenk.guendogan, peter.kietzmann, t.schmidt}@haw-hamburg).

O. Hahm was supported by INRIA and Freie Universität Berlin while working on RIOT (e-mail: oleg@riot-os.org).

M. Lenders, H. Petersen, M. Wählisch are with Freie Universität Berlin, Germany, (e-mail: {m.lenders, hauke.petersen, m.waehlisch}@fu-berlin.de).

K. Schleiser is a freelancer and was supported by INRIA, France, and Freie Universität Berlin, Germany (e-mail: kaspar@riot-os.org).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

Overview of this paper

II. Anatomy of an IoT Operating System

An operating system for IoT devices must pack a number of features including (but not limited to) hardware-abstraction, networking, and power management, in the presence of spartan hardware resource consumption. In this section, we survey in more details the requirements for an OS running on low-end IoT devices.

A. Low-end vs. High-end IoT Devices

Compared to high-end IoT devices such as smartphones and RaspberryPis, low-end IoT devices typically have a factor of 10^6 less memory, 10^3 less central processing unit (CPU) capacity, consume 10^3 less power, and use networks with 10^5 less throughput. Low-end IoT devices are based on three core components:

1) A micro-controller (MCU)—a single piece of hardware containing the CPU, a few kB of random access memory (RAM) and read-only memory (ROM), as well as its register-mapped peripherals.
2) Diverse external devices such as sensors, actuators, or storage, which are connected to the MCU via a variety of input/output (I/O) standards such as UART, SPI, or I2C [1].
3) One or more network interfaces connecting the device to the Internet, typically using a low-power transmission technology. Such transceivers can either be part of the MCU (this case is known as a system-on-chip (SoC)), or be connected as external device via an I/O bus.

Micro-controllers on low-end IoT devices differ substantially across vendors, even for the same CPU architecture. However, micro-controllers typically have in common that (i) they are single-core, with slow clock cycles in the order of few MHz, and (ii) they do not provide advanced features, such as an MMU.

B. OS Requirements for Low-end IoT Devices

Following from this environment, an OS for low-end IoT devices must strike a balance between several design directions to meet the requirements outlined below.

[1]respectively: Universal Asynchronous Receiver/Transmitter, Serial Peripheral Interface, and Inter-Integrated Circuit

1) Basic Performance Requirements: The key performance requirements for such an OS are (i) memory efficiency, (ii) energy efficiency, and (iii) reactivity. In order to fit the RAM and ROM budget on low-end IoT devices, the OS must achieve a small memory footprint. IoT devices are generally expected to last years on a single battery charge. An OS should hence provide built-in energy saving mechanisms exploiting as much as possible the low power modes available on IoT hardware. In several contexts such as sensing alarms or reacting to remote commands, IoT devices are expected to react in (near) real-time. An OS should thus be able to provide at least soft real-time capabilities.

Here, an important observation is that, depending on the approaches, improving performance on one aspect may impact performance in another aspect. For instance, some approaches to memory efficiency may yield more CPU operations and more copying, thus harming energy efficiency. Similarly, an approach using deep sleep modes to save more energy may incur indeterministic system restore delays, thus harming reactivity.

2) Network Interoperability Requirements: Low-end IoT devices are expected to be connected to the network. Link-layer technologies used in the IoT include various low-power wireless technologies such as IEEE 802.15.4, Bluetooth Low-Energy (BLE), or LoRa. Link-layers relevant in IoT also include wired technologies such as BACnet, Power-line Communication (PLC), Ethernet or Controller Area Network (CAN). An OS should thus offer support for heterogeneous radio and wired transceivers, as well as various link-layers. On top of that, seamless Internet connectivity is often expected from IoT devices. Hence, an OS should offer support for the standard low-power IP stack, including 6LoWPAN, IPv6, UDP, CoAP [38]. Support for additional protocols may also be required in order to route packets, manage devices, and interoperate in the Web of Things (WoT). For instance, additional protocol support may be needed to comply with distributed interoperability frameworks such as Thread and IoTivity.

Here, an important observation is that the requirements for network technology support are (i) heterogeneous and (ii) likely to evolve over time. For example, it should be easy to integrate upcoming experimental stacks (e.g. information-centric networking [20] or software-defined networking [34]). The OS should thus enable some level of modularity to facilitate system evolutions, and to easily fit various configurations – while still matching the basic performance requirements described in Section II-B1.

3) System Interoperability Requirements: Software on IoT devices may be complex, and must often comply to rigorous quality constraints. On the other hand, not only is currently available low-end IoT hardware diverse, but it also quickly evolves over time. Hence, non-portable IoT software should be minimal. At low level, an OS must provide hardware abstraction so that most of the code is reusable on all IoT hardware supported by the OS. At application and software library level, an OS should provide standard interfaces to plug in a wide variety of third-party software modules.

Here, an important observation is that employing exotic

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

programming models and languages may seem attractive to achieve performance requirements, but can fatally harm system interoperability requirements, by limiting the reuse of well-known development tools, and by reducing the pool of available programmers and libraries [36].

4) Security and Privacy Requirements: IoT deployments are expected to penetrate both private lives and industrial processes, raising high demands for security and privacy of systems and applications. Conversely, the sheer number of deployed IoT devices poses a severe threat to the general Internet infrastructure, as recently demonstrated by the Mirai botnet attack [21]. An OS for the IoT must be carefully developed and continuously reviewed to minimize its attack surface. Crypto primitives must be wisely selected and specifically adapted to meet the device constraints—typically unsuited to perform asymmetric cryptography at scale. Furthermore, to prevent unwanted privacy violations or data leaks, transparency is desirable concerning end-user data handling, with end-users remaining in control.

In that regard, decades of experience have shown that a good approach to provide trustworthy security standards is free open source software, under active development and continuous review by a grass-roots community. Moreover, fully transparent end-user data handling cannot be achieved without disclosing a full view on code. This can only be achieved with a truly open source approach.

III. RIOT Principles & Goals

RIOT is an open source OS, based on a modular architecture built around a minimalistic kernel, and developed by a world-wide community of developers. Before we detail the concepts at work in RIOT, it is important to note the goals which motivated the design of RIOT in the first place:

- minimized resource usage in terms of RAM, ROM, and power consumption;
- support for versatile configurations: 8-bit to 32-bit MCUs, wide range of boards and use-cases;
- minimized code duplication across configurations;
- portability of most of the code, across supported hardware;
- provide an easy-to-program software platform;
- provide real-time capabilities;

The above goals lead to a number of principles which explain parts of the design of RIOT. These principles are the following:

1) **Network Standards –** RIOT focusses on open, standard network protocol specifications, e.g. IETF protocols;
2) **System Standards –** RIOT aims to comply to relevant standards, e.g. the ANSI C standard (C99), to take full advantage of the largest pool of 3rd party software. Extensive use of C language caters for both low resource requirements and easy programability;
3) **Unified APIs –** RIOT aims to provide consistency of APIs across all supported hardware, even for hardware-accessing APIs, to cater for both code portability and minimize code duplication;

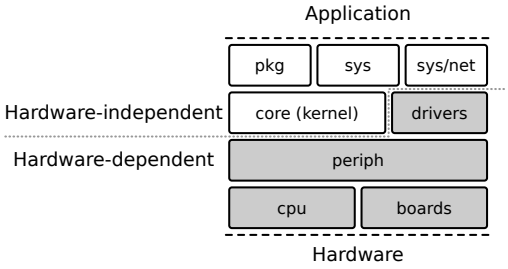

Fig. 1: Structural elements of RIOT.

4) **Modularity –** RIOT aims to define self-contained building blocks, to be combined in all thinkable ways, to cater both for versatile use cases and for memory constraints;
5) **Static Memory –** RIOT makes extensive use of pre-allocated structs to cater both for reliability, simplified validation and verification, as well as real-time requirements;
6) **Vendor & Technology Independence –** Vendor libraries are typically avoided, to preclude vendor lock-in and to minimize code duplication; Furthermore, design decisions should not tie RIOT to a particular technology;
7) **Open & Inclusive Open Source Community –** RIOT aims to remain free and open for everyone, and to aggregate a community with 100% transparent processes.

With such goals and principles in mind, the next sections describe RIOT in detail, following a bottom-up approach.

IV. The RIOT Software Structure

RIOT is structured in software modules that are aggregated at compile time, around a kernel providing minimalistic functionality. This approach allows to build the complete system in a modular manner, including only modules that are required by the use-case. This not only minimizes memory consumption, but also system complexity in the field. Still, modularity is balanced to avoid unmanageable structural convolution in the long run due to overly fine-grained software components [36].

At a high level, the RIOT code is structured according to the groups depicted in Fig. 1:

- `core` implements the kernel and its basic data structures such as linked lists, LIFOs, and ringbuffers;
- hardware abstraction distinguishes four parts: (i) `cpu` which implements functionalities related to the microcontroller, (ii) `boards` which mostly selects, configures, and maps the used CPU and drivers, (iii) `drivers` which implements device drivers, and (iv) `periph` which provides unified access to microcontroller peripherals and is used by device drivers;
- `sys` implements system libraries beyond the functionalities of the kernel, such as crypto functionalities, file system support and networking;
- `pkg` imports third-party components (libraries which are not included in the main code repository);
- `application` implements the high-level logic of the actual use-case.

Within code groups, functionalities are split into modules, each of which consists of one or more API header file(s)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

in the include path, and a single directory containing all its code. Modules may have sub-modules used for tailoring functionalities at a finer grain.

It is worth noting that the RIOT structure naturally factors out high-level application logic. Thus, a straightforward approach (and current best practice) is to develop and maintain high-level application code in a separate repository which typically pulls in the particular RIOT release the application is built upon.

The minimal configuration bundles only the `core` module (without any sub-modules), a `cpu` module and a `board` module. Every other module is optional. To give an idea, the minimal configuration requires 3.2 kBytes of ROM and 2.8 kBytes of RAM on 32-bit Cortex-M platforms (even less on 8-bit and 16-bit platforms). A configuration compliant with 6LoWPAN requires 38.5 kBytes of ROM and 10 kBytes of RAM. For more details and more examples of configurations see Section XIII.

V. The RIOT Kernel

The kernel of RIOT initially evolved from the FireKernel project [40]. It provides basic functionality for multi-threading: context switching, scheduling, inter-process communication (IPC), and synchronization primitives (mutex etc.). All other components, such as device drivers, network stack components, or application logic, are kept separate from the kernel as described in the previous section. Typically, interaction between such components is implemented via the minimalistic `core` API provided by the kernel.

A. Multi-Threading

A thread in RIOT is akin to a thread in Linux. Using the RIOT `core` API, each component – be it a driver for a network transceiver, or some application-specific logic – can run in a separate thread context with a thread priority level assigned to it. Multi-threading was built-in to provide the following advantages: (a) clean logical separation between multiple tasks, (b) simple prioritization between tasks and (c) easier import of code. Ignoring inherent additional complexity needed to manage concurrency, the most prominent drawback of multi-threading on low-end embedded devices is memory overhead. This overhead decomposes into (i) memory for the thread control block (TCB), (ii) memory for stack space, (iii) memory for CPU context (registers). While (iii) is determined by the CPU architecture (e.g. 64 bytes on Cortex-M, less on 16-bit and 8-bit), (i) and (ii) can be influenced by software. In RIOT, the TCB is designed to be very small, limiting the impact of (i). For instance, on Cortex-M, the TCB is 36 bytes in the default configuration and 12 bytes without messaging. A number of best practices (e.g., use of static memory allocation for data structures, no recursive functions) minimize the stack usage at run time. By combining the above it is possible to run threads with simple logic, starting from 128 bytes of RAM in total (on Cortex-M) for (i), (ii), and (iii).

a) Light-weight inter-process synchronization: The kernel provides various synchronization primitives such as mutex, semaphore and messaging (`msg`). These mechanisms are modeled as submodules of the kernel and are thus optional, and compiled only on demand. In terms of memory footprint, the size of these submodules is small, for instance ∼170 bytes of ROM for the mutex and ∼600 bytes of ROM for `msg` on Cortex-M. In terms of speed, the delay incurred by using IPC decomposes into the time for (i) saving and restoring thread contexts, (ii) the runtime of the scheduler and (iii) the runtime of the IPC submodule itself. While (i) is entirely determined by the CPU architecture, (ii) is constant as described in Section V-B, and a slim design of `msg` makes (iii) small overhead compared to (i) and (ii).

b) Multi-threading is optional: For some scenarios where extremely low memory usage is mandatory, a single-threaded application may be desirable. RIOT does not force the use of multiple application threads. Unless a selected system module needs to run a thread, the user application can be the only thread running on the system. In that case, it is possible to remove most of the memory requirements of the scheduler. This way, it is possible to build extremely memory-efficient firmwares in an Arduino-like fashion, still benefiting from RIOT's features including hardware abstraction, device drivers, and tooling.

B. Scheduling & Real-Time Properties

The kernel of RIOT uses a scheduler based on fixed priorities and preemption with $O(1)$ operations, allowing for soft real-time capabilities [31]. In more detail: the time needed to interrupt and switch to a different thread will not exceed a (small) upper bound, since context saving, finding the next thread to run, and context restoring are all deterministic operations. A class-based run-to-completion scheduling policy is used: the highest priority (active) thread runs, only interrupted by interrupt service routines (ISRs). With this scheduler, RIOT thus provides a clean way to prioritize tasks and preempt handling of low priority tasks in order to deal with high-priority events.

The scheduling policy used in RIOT simplifies real-time scheduling in that, if an event requires action by a high priority thread, lower priority threads are preempted and the high-priority thread runs until the event has been handled. Note that, in order to minimize processing time and energy consumption, there are no context switches mimicking parallel execution of tasks of the same priority. These characteristics allow for deterministic, real-time system behavior – provided task priorities are configured coherently. Furthermore, these characteristics also support well use-cases with mixed-criticality, whereby real-time tasks run alongside best effort tasks – e.g. an engine control task (real-time) running on the same board with an IP networking stack (non-real-time).

The scheduler used in RIOT is *tickless*. It does not depend on CPU time slices and periodic system timer ticks. The system thus does not need to periodically wake up unless something is actually happening, e.g. an interrupt triggered by connected hardware. A wake-up may be initiated by a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

transceiver when a packet has arrived, by timers when they fire, by buttons being pressed, or similar. If no other thread is in running state and no interrupt is pending, the system switches by default to the idle thread – which has lowest priority. The idle thread in turn switches to the most energy-saving mode possible (see Section VIII), thus optimizing energy consumption.

VI. Hardware Abstraction

Commissioning and managing hardware resources are central aspects for an OS. In that regard, RIOT aims to provide comprehensive hardware abstraction.

Hardware targeted by RIOT typically decomposes in a micro-controller including its peripheral units (MCU), and a number of external components such as sensors, actuators, and network interfaces (see Section II for details). These components are either connected via one or more printed circuit boards (PCB), or gathered in a system on chips (SoC) which combines an MCU and other elements on the same die. From here on in this paper, we will use the term CPU is used as synonym for MCU, meaning both the CPU core and its peripherals which are register-mapped.

The hardware abstraction in RIOT reflects this composition by structuring all hardware-dependent code into three blocks, stored in the folders `boards`, `cpu`, and `drivers`. Great care is taken to prevent code duplication as much as possible. In general, hardware abstraction in RIOT is designed to speed up porting efforts, by capitalizing automatically on existing code. If the CPU and drivers are already supported, porting RIOT to a new board is simply a matter of creating a few configuration files (this can be done under an hour by a skilled developer). Otherwise, the complexity of supporting a new CPU, or a new driver, depends obviously on the actual hardware. However, on one hand, new code which may be needed is limited to `boards`, `cpu`, and `drivers`. And on the other hand, existing code outside of these folders (see Fig. 1) works out-of-the-box.

Every RIOT build includes exactly one instance of a `board`, one instance of a `cpu` implementation, and zero or more `drivers`. Tasks addressed by each of these three blocks and details of their implementation are described in the next sections.

A. CPU Abstraction

The `cpu` abstraction gathers implementation and configuration of all aspects of the micro-controller in use. This includes specific code for handling interrupts, context switching, system clock management, timers and drivers for peripherals such as UART and SPI.

Core abstraction – RIOT follows a hierarchical approach distinguishing the CPU architecture (e.g. Cortex-M4), the CPU family (e.g. `stm32`), the CPU type (e.g. `stm32f4`), and finally the actual CPU model (e.g. `stm32f446re`). This reflects the way different CPU vendors structure their portfolios. An example for the Cortex-M CPU architecture is depicted in Fig. 2.

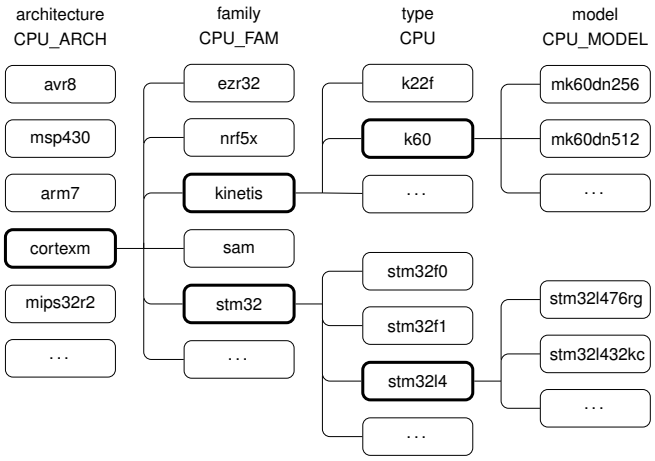

Fig. 2: Hierarchical code organization of CPUs in RIOT.

In general, all CPU-dependent implementation artifacts are placed as close as possible to the root of the tree. Referring to our previous example, assembly code for context switching and interrupt handling for Cortex-M is implemented once, for all Cortex-M CPUs, and is reused by all CPUs based on this architecture – thereby reducing the effort of porting new Cortex-M CPUs. For efficiency reasons, this structure is not enforced: tailoring is tolerated to cater for the specific properties a particular platform may have. For instance, the Linux-based CPU abstraction used for the *native* port (see Section XIV) does not need such hierarchical structure and is thus implemented in a single module.

Generic Peripheral API – RIOT provides vendor- and architecture-agnostic APIs for typical micro-controller peripherals (`gpio`, `uart`, `spi`, `pwm` etc.). The goal of these interfaces is to increase code portability by providing unified, but direct and fine-grained access to such peripherals. For example, through this API, using an SPI bus on a 8-bit Atmel megaAVR CPU is identical to using an SPI bus on a 32-bit ST Cortex-M4. Note however that the intention here is not to model into the API all possible (often vendor-specific) operation modes, but rather to provide the basic functionality which covers most use cases.

In general, these peripheral interfaces are implemented from scratch, directly on top of the register definitions, without using any vendor libraries. This has proven to be much more efficient in terms of run-time overhead and code size (e.g. $\sim$200 lines of code for the STM32F4's SPI driver in RIOT vs. $>$1500 lines for the driver in the vendor library). However, this convention is not strictly enforced, as the decision to use vendor libraries is made per CPU family.

B. Board Abstraction

A `board` in RIOT is considered as the software counterpart to a PCB in hardware. The term `board` in this context thus refers to an IoT device as a whole, and is used as synonym with hardware target, or platform. The board maps (i) the selection and configuration of the CPU in use, (ii) the selection and configuration of drivers for components available on-board,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

(iii) board-specific initialization code, and (iv) the tooling and its configuration needed to interact with the board, e.g. to program it.

For the CPU in use, the board specifies the full configuration of CPU peripherals, including not only the number of UART, SPI, or timers used, but also their pin mapping, clock settings and related aspects. The board configuration also selects the driver modules that should be used and specifies their configuration details, including the number of active instances, the mapping of those instances to pins and CPU peripherals, providing information on operating modes. Some boards need to setup and control external pin multiplexers, power management chips, etc. In such cases, the specific implementations provide the necessary start-up and management code. Settings for programming and debugging the IoT device including the selection of the flashing tool (e.g. `OpenOCD`), the default serial port, etc, are also part of the configuration.

C. Driver Model

Drivers in RIOT are software modules that control CPU-external components such as sensors, actuators, storage, or network transceivers. The driver model is based on three major design objectives: (i) driver implementation should be independent from the board and CPU, (ii) drivers should allow for multiple active instances, and (iii) drivers should implement common high-level APIs. Sticking with these objectives allows not only for the drivers to be re-used on any board, but also to swap components and drivers without changing user application code.

Drivers typically control hardware components connected to CPU through general purpose pins (GPIO) and buses such as UART, SPI, or I2C. To be able to communicate with the target component, a driver thus needs to access the corresponding micro-controller peripherals. A driver achieves this through RIOT's **peripheral API** provided by the CPU implementations (see Section VI-A). By enforcing this abstraction, all drivers are CPU-agnostic and can be used with any supported board. Furthermore, drivers do not need to be modified when porting new boards to RIOT.

It may be required by applications to use several components of the same type. For example, a robot might use multiple distance sensors, or an IoT device might use two temperature sensors to measure simultaneously temperature in-room and outside. For such cases, driver implementation must allow multiple active instances of the same driver. The typical reason why this may be difficult, is that drivers generally keep their state in a set of global variables. The driver model in RIOT solves this issue by requiring drivers to bundle all their run-time state into a single, separate data structure. The pointer to this structure is called the *device descriptor* and is used to identify a specific device instance.

Generic Driver APIs – To enable code portability on top of component drivers, a common high-level abstraction is needed. For instance, it is desirable to use a 6LoWPAN network stack *unchanged* on top of various radio transceivers. In the RIOT driver model, this is achieved by categorizing hardware components into classes and by providing a dedicated high-level API for each class of hardware component. Currently, three APIs are defined: `netdev` for network transceivers, *SAUL* for sensors and actuators, and *MTD* for (flash-) storage components. While the MTD interface is modeled after the corresponding Linux API, the other two interfaces are designed specifically for RIOT. The next section describes SAUL in more detail, and `netdev` is described in Section X.

D. Sensor/Actuator Abstraction

RIOT provides a generic API for accessing sensor and actuator devices, called the **Sensor Actuator Uber Layer API, SAUL**. This API allows not only a vendor-agnostic access to sensors and actuators, but also allows to write applications against heterogeneous IoT devices, using the same function calls.

SAUL builds upon *device descriptors* defined by RIOT's driver model (see Section VI-C) and upon a generic data type called `phydat`. Sensors and actuators work with physical values, e.g. temperature sensors read temperatures in °C or °F, motors are controlled setting the torque (Nm). Based on this observation, RIOT defines a self-describing and compact (8 bytes) data type encoding these physical values. This data type contains up to 3 numerical values, combined with a scaling factor and an enumeration value specifying the physical unit in use. `phydat` thus uses a custom floating point notation with a 15-bit mantissa for each value and a shared 8-bit exponent – providing a precision sufficient for interfacing with all common sensors and actuators. Although these values allow for simple conversion into regular floating point numbers on a higher level, the CPU does not need to know about floating point arithmetics when just handling the `phydat` structure. To read or write values from/to a target sensor or actuator, SAUL requires only the descriptor of the target device and a pointer to a `phydat` structure.

On top of the SAUL interface, RIOT provides a device registry for sensor and actuator devices. This registry enables iterating through all SAUL-enabled devices at run-time, which allows applications to automatically adapt to available devices. While this can be used as base for plug-and-play-like behavior, the typical use case is to iterate through the SAUL registry for auto-configuration during system initialization. For instance, the SAUL registry can be used to automatically map all SAUL-enabled devices to CoAP resources, thus allowing for remote access to all local sensors and actuators.

VII. System Booting

RIOT provides all elements for bootstrapping IoT hardware from the very first software instruction called (typically some kind of reset interrupt handler) up to the point where the actual `main` function is called in a thread context. Wherever possible, all initial initialization code is implemented in plain C, which leads to a better level of maintainability than the typical approach based on assembly code. The system start-up sequence consists typically of the following steps: (i) memory bootstrapping, (ii) initialization of board and CPU, (iii) initialization of the C-library used (this step is optional), and finally (iv) setup and initialization of the actual operating system.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

The memory initialization simply takes care of copying the initialized variables into RAM (`data` section) and setting all uninitialized variables to zero (`bss` section). This step can optionally be extended with hardware specific sub-tasks, e.g. application of fixes for hardware bugs, as specified in the vendor errata sheets.

The board initialization then takes care of initializing board specific hardware elements, such as on-board LEDs, I/O-expanders, or CPU-external power management devices. At this point, the CPU initialization is also triggered, which takes care of setting up the interrupt system, the clock tree, shared peripheral drivers etc.

Next, the C-library initialization takes care of mapping *libc* functions (e.g. functions for dynamic memory allocation or `stdio` access) to the corresponding RIOT syscalls.

At this stage the base system is initialized, and control is handed over to the RIOT kernel. After setting up the `idle` and `main` threads and switching into the latter, RIOT initializes all configured system modules and device drivers through its `auto_init` module. Lastly, the actual `main` function is called, which is the entry point for the actual user application.

It is worth pointing out that the described start-up sequence can be tailored in various ways. For example, applications can be designed to call all device driver and system module initializations manually, by de-selecting the `auto_init` module.

VIII. Power Management

To design power management, it is crucial to first understand both the source(s) of energy consumption, and the influence software can have on these. A device's total energy consumption is the sum of the power consumed by (i) the CPU, (ii) devices connected through peripherals, and (iii) various other passive components external to the CPU. While (iii) is heavily influenced by a board's hardware design, both (i) and (ii) are influenced by software.

On one hand, the power management system in RIOT automatically sets the CPU in the deepest possible sleep mode when idle, which decreases energy consumption due to (i). On the other hand, user applications or other system modules are expected to deal with (ii) by managing the state of peripherals. For example, a network transceiver is managed by a MAC layer protocol module, while a sensor is managed by a sensing application.

The core design element in the CPU's power management system in RIOT is the so-called `idle` thread. This thread is created during system startup and is scheduled when no other thread needs to run, i.e. when the CPU is idle. The CPU's power management is built around a single function, which triggers the CPU to go into the lowest possible power state (sleep mode). Calling this function is the only task the `idle` thread performs. This mechanism works transparently for user applications and other system modules, as follows.

A module called `pm_layered` implements the default mechanism which determines the lowest possible power state. This module is shared by various CPU implementations. The `pm_layered` module uses a simple consensus mechanism called *Cascade*, distributed between CPU peripherals and user threads. *Cascade* is based on a strict hierarchy of power mode levels, that are defined on a per-CPU basis. A lower power level means less power consumption. The hierarchy is such that, if power level N is blocked, all power levels $M \leq N$ are implicitly also blocked, thus preventing the CPU to switch to any power level $M \leq N$. When using the `pm_layered` module, peripheral drivers or application code can each independently (un)block power modes that are (in)appropriate to run on. For example, a UART driver can block all deep sleep CPU states, as these would prevent the UART peripheral to work correctly. The UART driver would keep the deep sleep power state blocked while the UART peripheral is enabled, and unblock the power state once the peripheral is disabled. Independently, when scheduled, the `idle` thread switches the CPU to the lowest power mode currently unblocked in `pm_layered`.

Note, however, that using the `pm_layered` module is not mandatory. CPU implementations can be tailored to use a mechanism other than *Cascade* for determining the lowest possible power state, e.g. via hardware.

IX. Timer Subsystem

Timers are a very important part of any operating system, as they provide a mapping between the physical time as perceived in the outside world, and the internal timing used by the CPU (i.e. ticks). A typical challenge with timers is dealing with high dynamic range, covering time spans from nanoseconds up to days and months. Another challenge is the (potentially large) number of timers required to run in parallel depending on the application and the situation, while a hardware platform only provides a limited, constant number of timers.

Typical CPUs supported by RIOT provide a set of general purpose `timers`, and optionally one or more real-time timers (RTTs), or a real-time clock (RTC). Each of these provides a number of capture-compare channels which determine the number of timers that can be set at the same time.

To make these low-level timers available to application developers in a transparent way, a high-level timer subsystem is needed. As a minimum, this subsystem needs to provide three capabilities: (i) tick conversion, (ii) multiplexing, and (iii) range extension. *Tick conversion* simply refers to the ability to convert physical units to timer ticks. On a system with a 2 MHz timer, for example, one millisecond is mapped to 2000 ticks. *Multiplexing* refers to the subsystem's ability to map any number of active timers onto a fixed number of available hardware timer channels. Lastly, *range extension* enables the timer subsystem to handle timers that exceed the range of the underlying hardware timer, e.g. for setting a 1 s timer on a 16-bit hardware timer at 1 MHz, which overflows every 65 ms.

RIOT provides such a high-level timer subsystem, called `xtimer`, which implements these three capabilities. `xtimer` offers a simple API based on natural time values which provides full abstraction from the underlying timer hardware – means for putting threads to sleep, setting callback and event timers etc.

For interfacing with actual hardware timers and RTCs/RTTs, `xtimer` uses the peripheral driver interface. This way the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

implementation is completely hardware-independent while still being configurable to work with various hardware environments. In order to multiplex on top of the configured hardware timers, `xtimer` data structures are stored in lists. Note that the $O(n)$ complexity for setting and removing timers implies that the overall number of timers used in a specific system configuration needs to be taken into account when analyzing the system's real-time behavior. Internally, `xtimer` uses a configurable 64-bit time base, typically in a microsecond granularity. Due to this, the subsystem can cope internally with a very high dynamic range, and can – in conjunction with overflow counters – cope with any type of underlying hardware timer.

`xtimer` thus provides a single, portable timer API to RIOT which can scale in magnitude, from fine granular timings in microsecond up to tasks scheduled in hours or days.

X. Networking Subsystem

The networking subsystem spans from transceivers to application logic and is an integral part of any IoT operating system. RIOT defines a flexible, layer-separating architecture [35] and two programming interfaces for interacting with the network subsystem. Southbound, the `netdev` interface offers a generic network transceiver driver interface. Northbound, the `sock` interface provides high-level network access for applications (in equivalence to POSIX sockets). Examples of network subsystems harnessed by RIOT via these interfaces include the default IP stack of RIOT (named GNRC [35]) and a variety of other IP protocols stacks (see Section XIII), as well as experimental protocol implementations such as a content-centric networking subsystem as shown in [30], [33].

A. The netdev Interface

`netdev` handles (i) data sent and received by the transceiver, (ii) transceiver configuration and initialization, and (iii) transceiver-related events. This interface is generic in the sense that a network subsystem accessing the transceiver through `netdev` can run absolutely unchanged if the transceiver is replaced by another transceiver using a similar link-layer technology.

1) Sending and receiving: `netdev` defines a single, asynchronous function for sending data to the network device driver. Data is passed with a `struct iovec` structure, and the `send` function typically returns as soon as the data has been copied into the internal transmit buffer of the transceiver. However, synchronizing on actual transmission events such as *transmission started* or *transmission completed* can be configured to trigger corresponding events, as described below.

`netdev` also defines a single function for receiving data from the network driver. As soon as the driver triggers an event to indicate that the reception of a packet has been completed (`RX_COMPLETE`), received data can be read using the `recv` function which allows to either (a) obtain the size of the incoming data, (b) read the incoming packet into a given buffer and drop the packet afterwards, or (c) drop the incoming data without reading it. The `recv` function can also be used to pass cross-layer data such as LQI, RSSI etc.

2) Transceiver configuration & initialization: `netdev` provides generic `get` and `set` functions that reads or writes network driver configuration options in a key-value syntax. Typical keys are link layer address, radio channels, or transceiver states. All options are listed and identified in an extensible enumeration type called `netopt`. It is up to a device driver to service a given option, or to simply return an error code in case the option is not supported. Finally, `netdev` also exposes a generic `init` function to initialize a transceiver.

3) Transceiver event handling: `netdev` handles events that fall into the four categories: receive (RX), transmit (TX), link, and system.

RX and TX events cover all notifications that are triggered in the process of sending and receiving data through the network. Most prominently, transceiver drivers support the `RX_COMPLETE` event that is triggered every time new data is available. A variety of other events such as link layer state changes are supported by a subset of transceiver drivers. Link events allow a transceiver driver to signal changes of the link state or link configuration to the upper network layers and are useful to initiate operations at the network layer. For instance, a `LINK_UP` event is typically used to start an IPv6 router solicitation. System events allow to notify the thread running the corresponding transceiver driver from an ISR. In conjunction with an `event_callback`, this allows programmers to specify how to call the event handler of a driver.

B. The sock Interface

`sock` is a collection of high-level network access APIs similar to the POSIX socket API. These interfaces are generic in the sense that an application accessing the network through `sock` can run absolutely unchanged if the network subsystem is replaced by another network stack.

In contrast to POSIX sockets, `sock` interfaces do not require any dynamic memory allocation or internal state maintenance. The state variable for an end point is always provided via its function call—an approach somewhat similar to `netconn` from *lwIP* [26]. However, in contrast to the latter or to POSIX sockets, `sock` is an API collection for various types of transport protocols or channels. Users can thus keep their implementation lean by confining it to those parts of `sock` that are actually needed for their specific use case.

Currently supported are `sock_udp` for UDP traffic, `sock_ip` for raw IP traffic, and `sock_tcp` for TCP traffic, each of which provides means to:

- *create* end-points for the respective communication type,
- *send* data via those end-points or group addresses,
- *receive* data in either blocking mode (with optional time-out) or non-blocking mode, and
- operate *protocol-specific actions*, e.g., connecting to or listening for another peer in TCP.

Portability is ensured by conventions using only common types and definitions, either from *libc* or POSIX. By combining these elements, `sock` enables easy porting of applications between RIOT and other operating systems. For instance,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

Package	Overall Diff Size	Relative Diff Size
ccn-lite	517 lines	1.6 %
libfixmath	34 lines	0.2 %
lwip	767 lines	1.3 %
micro-ecc	14 lines	0.8 %
spiffs	284 lines	5.5 %
tweetnacl	33 lines	3.3 %
u8g2	421 lines	0.3 %

TABLE I: Some RIOT packages: # lines of patch files vs overall LoC of the library.

RIOT itself provides the POSIX socket API as a wrapper around `sock`, which is used to port external libraries to RIOT (see Section XI). Conversely, `sock` can easily be wrapped around a POSIX socket, thus making applications developed against `sock` portable to any POSIX-based operating system.

XI. External Libraries

RIOT caters for integration of third-party software and libraries as *packages* (in the following referred to as `pkg`), following an approach similar to BSD ports [5]. At compile time, the `pkg` system uses a Makefile template which specifies (i) how to automatically download a certain version of the software, e.g. from an upstream code repository, (ii) how to integrate the library seamlessly into RIOT, and optionally (iii) , a collection of patch files and/or glue code adapting this library to RIOT. Typically, patch files are minimalistic, such that a package is reduced to a RIOT-specific Makefile and minor code adaptation to comply with RIOT's rather strict compiler settings (see Table I). Otherwise, a `pkg` is equivalent to a module (see Section IV) from the point of view of the RIOT build system. The `pkg` system thus allows convenient and transparent integration of external libraries into RIOT with minor effort. Note however that some packages need more adaptation, e.g. to make them hardware-independent as expected by RIOT.

A wide variety of libraries are available as RIOT packages, ranging from system libraries such as the TLSF malloc implementation, to u8g2 for basic graphics; from crypto libraries providing strong security primitives such as tweetNaCl or micro-ecc, to full network stacks such as lwIP or OpenThread, providing network-level interoperability.

Note that packages are typically integrated via the generic system-level APIs defined by RIOT. For instance, packages importing full network stacks in RIOT make use of `netdev` and `sock` interfaces (see Section X) so that it is possible to both (i) replace the network subsystem without any change to drivers or to application code, and (ii) use these libraries on top of most of the hardware supported by RIOT.

XII. RIOT Application Programming Interfaces

This section recaps the main application programming interfaces defined by RIOT. It is important to note that these interfaces are consistent across all hardware supported by RIOT, meaning that coding against such interfaces guarantees cross-platform code portability. See Table II, which lists the APIs, as well as Fig. 3, which shows how these APIs articulate. The APIs fall into four categories:

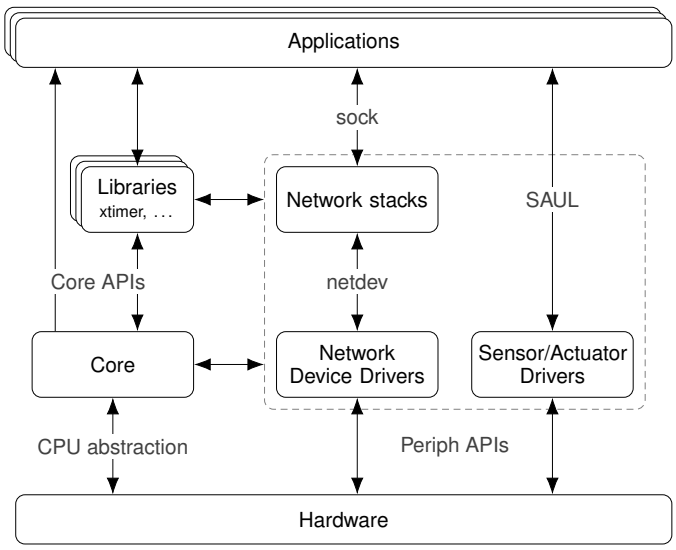

Fig. 3: The main programming APIs in RIOT.

Core APIs provide basic OS kernel interfaces to manage threads, interprocess communication, and concurrency.

Hardware abstraction APIs provide generic interfaces for abstracting MCU peripherals as well as unified access to device drivers grouped by device classes;

Timer APIs provide unified high-level access to the configured, platform dependent timer infrastructure;

Network APIs provide a generic interface to different network stacks.

XIII. Example Configurations

This section gives an overview of the memory usage for a number of typical build configurations. All of the numbers given in this section are produced by building the below configurations for the Atmel `samr21-xpro` board, which is COTS hardware. The `samr21-xpro` integrates a 32-bit Cortex-M0+ and an IEEE 802.15.4 radio on the same die. Compared to the measurements reported in the below sections, RIOT's memory footprint on other hardware is either similar (for other Cortex-M boards) or smaller (for 8-bit and 16-bit hardware).

Every application is compiled with `arm-none-eabi-gcc` version `5.4.1`, and built without debug information (no `DEVELHELP`, shell etc) using the default configuration unless stated otherwise. In the following sections, we use the notation of $[X/Y]$ kB to denote the usage of X kilobytes in ROM (`.text` + `.data` sections) and Y kilobytes in RAM (`.bss` + `.data` sections). The `.data` section is counted in both ROM and RAM usage, as its content is saved into the flash and moved into RAM upon booting the system. All values are rounded to 100 byte precision: the aim here is not to claim measurement precision up to the last byte, but rather to provide an idea of the memory requirements.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

Core APIs	Hardware Abstraction APIs	Timer API	High-level Network APIs
`thread` thread handling	`periph/uart, spi, ...` access to MCU peripherals	`xtimer` high-level timer abstraction	`sock` generic interface to high-level connectivity
`msg, mbox, thread_flags` inter-process comm.	`SAUL` access to sensors/actuators		`POSIX socket` based on sock
`mutex, semaphore` concurrency handling	`netdev` access to network devices		
	`mtd` access to storage devices		

TABLE II: Overview of common system APIs to implement applications in RIOT.

RIOT Configuration	ROM	RAM
Basic RTOS	3.2 kB	2.8 kB
6LoWPAN-enabled	38.5 kB	10.0 kB
JavaScript-enabled	166.2 kB	29.1 kB
OTA-enabled	111 kB	17.5 kB

TABLE III: Measurements of RAM and ROM for various configurations of RIOT, on typical IoT hardware based on a 32-bit Cortex-M0+ microcontroller.

A. Basic RTOS Configurations

A barebone configuration with just the kernel running on top of the hardware abstraction (`cpu` and `board`, see Section VI) has a memory footprint of $[3.2/2.8]$ kB, of which 2.2kB are stack space. This configuration is reproducible by building `tests/minimal` in the RIOT codebase. A basic hello-world configuration, adding standard I/O, has a memory footprint of $[7.9/2.9]$ kB. This configuration is reproducible by building `examples/hello-world` in the RIOT codebase.

B. Configurations with IP Network Interoperability

Several configurations can provide IP network interoperability. A simple configuration providing IPv6, 6LoWPAN and CoAP interoperability with the RIOT default network stack (`GNRC` [35]) has a memory footprint of $[38.5/10.0]$ kB. This configuration is reproducible by building `examples/gnrc_minimal` and adding the `gcoap` module. Alternative configurations can provide the same network interoperability, but integrate other network stacks supported by RIOT. For instance, a matching configuration using the `lwIP` [26] network stack has a memory footprint of $[52.7/15.8]$ kB, while a matching configuration using the `emb6` [32] network stack has a memory footprint of $[39.5/10.2]$ kB.

C. Configuration with JavaScript Interpreter

A configuration enabling high-level application programming in standard JavaScript (full EcmaScript 5.1 compliance), with the interpreter running directly on the device has a memory footprint of $[166.2/29.1]$ kB. Note that $[131.0/26.8]$ kB of this is used by the JavaScript engine. This configuration is reproducible by building (`examples/javascript`) in the RIOT codebase. An alternative configuration with similar memory footprint has been tested in [22] whereby the main APIs of RIOT are mapped in JavaScript, and a small JavaScript runtime container is exposed as a web resource (via CoAP).

D. Configuration with Secure Firmware Updates Over-the-Air

Several configurations can provide over-the-air firmware updates. A typical configuration assembles (i) a small bootloader, and (ii) two firmware slots, similar to the approach of MCUboot [7]. A possible configuration uses for (i) an RTOS configuration of RIOT similar to the one described in Section XIII-A. For each slot in (ii) a RIOT image featuring a software update module can be used by building upon one of the 6LoWPAN network stack configurations described in Section XIII-B. The total memory footprint of this composite configuration is roughly $[111/17.5]$ kB, consisting of $[3.1/0.8]$ kB for (i) and $[53.7/17.5]$ kB for each of the OTA-enabled images in (ii). Within this memory footprint, the software update module provides the necessary functionalities to periodically poll remote software update servers through the IP network, download firmware updates over-the air, and verify integrity and authenticity of the downloaded firmware using state-of-the-art public-key crypto and hashing, before booting the new firmware.

XIV. Tools & Code Quality Workflow

The default development process for RIOT is based on a set of standard open source programs, including tools for building (*gcc*, *make*), static analysis (*cppcheck*, *coccinelle*), dynamic analysis (*Valgrind*), network sniffing (*Wireshark*), standard debugger (*gdb*), performance profilers (*gprof*), unit testing (*embUnit*) and standard flashing tools (*OpenOCD*).

A. Debugging and Testing Tools

1) Shell: RIOT provides a *command-line interpreter (CLI)* similar to a shell in Linux. This *CLI* is typically accessed (remotely) over UART. It is designed to facilitate debugging and run-time configuration while testing or conducting experiments. Developers can easily add custom commands through dedicated command handlers.

2) RIOT native: A basic *hardware virtualizer* allows the compilation and execution of RIOT applications as user processes in a host OS (supported host operating systems include Linux, FreeBSD, and Mac OS X). This virtualizer, called RIOT *native*, provides a basic emulator for typical IoT devices: a board and CPU featuring timers, a UART, a network interface, as well as basic sensors and actuators. Based on RIOT

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2815038, IEEE Internet of Things Journal

native, up to hundreds of RIOT instances can run in parallel on the host OS. RIOT *native* instances can communicate with one another, or with the Internet, through virtual Ethernet interfaces (using TAP interfaces on the host system). Going further, RIOT can also be compiled and tested with advanced emulators of relevant IoT hardware, such as MSPsim & Cooja (emulating TI MSP430), AVRSim (emulating Atmel AVR), and Renode (emulating ARM Cortex M).

B. Standard Workflow

A standard RIOT development methodology was defined [11] which aims to shorten time-to-running-code on low-end IoT devices. This methodology enables developers to efficiently harness the aforementioned ecosystem of open source software development tools. The methodology distinguishes three phases whereby a developer (i) writes, debugs, and tests the developed IoT software on RIOT *native*, then (ii) remotely runs and debugs the software on real hardware by using open-access *IoT testbeds* supported by RIOT, such as IoT-LAB [15] [19], before finally (iii) running the code on the targeted IoT device. This approach provides a controlled environment to develop IoT software which speeds up development and lowers the bar of entry for developers, on one hand by being standard, free and open source, and on the other hand by not requiring any specific IoT hardware on the premises, for phases (i) and (ii).

C. Code Quality Management

The RIOT community specifies well-defined processes that aim to ensure high code quality and solid documentation. The processes leverage state-of-the-art open tools for source code management (git and GitHub [10]) and documentation (Doxygen, wiki). Strict and precise coding conventions favor uniform code style and clarity, based on C99 (officially: *ISO/IEC 9899:1999*), Linux kernel coding style, systematic use of source code beautifiers such as *uncrustify*, as well as common boiler plates with license and authorship information. Systematic use of Doxygen comments for API documentation is enforced by mandatory checks in the Continuous Integration (CI) testing. This way, up-to-date documentation can be automatically exposed on the Web [9].

The review of code contributions to the master branch follows a defined, transparent pull-request process on GitHub. Open, back-traceable discussions on proposed code lead to consensus-driven merge decisions, moderated by a pool of RIOT maintainers (see Section XV). In particular, the code review process prior to merging enforces a clean and manageable commit history on the master branch.

Last but not least, the code review process mandates advanced CI testing based on a framework called *Murdock* [8]. *Murdock* CI slaves are distributed over multiple sites in Europe which have volunteered computing power. *Murdock* includes a frontend which is conveniently integrated into GitHub, as well as exposed on the web [8]. In the final stages before being merged in the master branch, a pull-request is tested via *Murdock*, which automates a large number of static tests (including *cppcheck* and *coccinelle* tests), as well as unit tests and compile tests for over 15000 build configurations, and functional tests on selected platforms.

All in all, the code quality management processes aim to scale by making the best of (i) distributed computing power volunteered by participants in the RIOT community to provide CI testing nodes, (ii) advanced automated testing and extensive up-to-date documentation, and more generally (iii) pooled IoT programming skills amongst RIOT developers and maintainers.

XV. The RIOT Open Source Community

The RIOT community gathers a large number of open source code contributors [3] from around the world: at the time of writing, the master branch of RIOT gathers the contributions of more than 170 developers from industry, academia and the maker/tinkerer communities. Both large companies (e.g. Cisco, Continental, Samsung) and SMEs (e.g. Eistec, Hamilton) contribute code, while various other companies (e.g. Fujitsu, Atmel, Nordic, Eclipse Foundation) sponsor or support related activities and events, such as the annual RIOT Summit [16].

The grass-roots RIOT community formalized a set of open processes [13] aiming at organizational durability, vendor-independence and transparency. Governance is mainly driven by RIOT maintainers (a status obtained via meritocracy) who have code review duties and merge rights on the master branch. Processes include periodical (virtual and f2f) meetings amongst developers and maintainers, partly inspired by organizational aspects of IETF and Linux communities which have proven scalable and durable.

Going one step further, the RIOT community also analyzed the non-trivial question of the license of the code [12], which is key to foster long-term community coherence and fruitful interaction with the industry. The result of this analysis confirmed RIOT's choice of a non-viral copyleft license (LGPLv2.1), while user applications, external libraries and packages can be based on other open source licenses (or be closed source). The RIOT community consensus shaped a strong belief that such a license provides the most appropriate framework to simultaneously (i) favor end-user protection by ensuring a durably free, open source, and up-to-date code base in RIOT's master branch, and (ii) foster business models around this free core, i.e. indirect business models à la Linux.

Various products shipped with RIOT have appeared since 2017, such as environmental sensors bundled with cloud back-end services [6], smart heating devices [4], smart automotive devices [14] or low-cost low-power communication modules [17]. Simultaneously, since 2013, hundreds of academic works in the field of IoT research have been based on RIOT, or reference RIOT.

XVI. Conclusion & Perspectives

This paper presented a comprehensive overview of RIOT, an open source operating system for low-end embedded devices, around which a large community of developers has recently aggregated. The technical contributions of RIOT, its architecture and core implementations, are discussed along with

community aspects—the organization of and experiences with the large open source ecosystem around RIOT:

Contrary to other prominent operating systems in the domain, RIOT takes an approach purposely similar to the GNU philosophy of Linux in terms of code license, vendor-independence and transparency. From a technical perspective, though, RIOT is written from scratch (without using vendor libraries) and differs from Linux in terms of the OS architecture.

In the current context, tension is mounting between large business requirements on the one hand, and individual privacy and security requirements on the other. Smart sensing and the Internet of Things are expected to exacerbate this tension even more. As shown by Linux, using a free open source software platform can strike a good balance between protecting end-users and supporting industry, in the long run.

As a whole, these characteristics aim at sustaining code performance and grassroots community coherence. However, some characteristics of RIOT (e.g. code license, no use of vendor libraries) have a potential to slow initial progress. Whether or not this tradeoff will make RIOT fulfill its promises of success similar to Linux remains to be seen. Nevertheless, the availability of free open source ecosystems such as RIOT will be crucial in democratizing the Internet of Things, and for this is worth the effort and extra time.

Availability

RIOT is open source, and publicly available on GitHub at

`https://github.com/RIOT-OS/RIOT`

Acknowledgements

Thanks to the whole RIOT community for the tremendous work. Thanks to Alexandre Abadie, Joakim Nohlgard, Daniel Petry and Bas Stottelaar for feedback on the text of this paper. Thanks to Peter Schmerzl for fundamental inspiration, and to Heiko Will for having been under fire. We also would like to thank the public research institutions which co-founded RIOT: Freie Universität Berlin, INRIA, and Hamburg University of Applied Sciences.

References

[1] Arduino. http://arduino.cc/.
[2] ARM mbed OS. https://mbed.org/technology/os/.
[3] BlackDuck analysis of RIOT. https://www.openhub.net/p/RIOT-OS.
[4] Eisox Smart Heating. https://www.eisox.fr.
[5] FreeBSD Ports. https://www.freebsd.org/ports/.
[6] Hamilton IoT. https://hamiltoniot.com.
[7] MCUboot. https://runtimeco.github.io/mcuboot/.
[8] Murdock: RIOT Continuous Integration. https://ci.riot-os.org.
[9] RIOT Application Programming Interfaces. https://api.riot-os.org/modules.html.
[10] RIOT Code-Base. https://github.com/RIOT-OS/RIOT.
[11] RIOT Coding Best Practice. https://github.com/RIOT-OS/RIOT/wiki/Best-Practice-for-RIOT-Programming.
[12] RIOT Community License Discussion. https://github.com/RIOT-OS/RIOT/wiki/FAQ.
[13] RIOT Community Processes. https://github.com/RIOT-OS/RIOT/wiki/RIOT-Community-Processes.
[14] Sleeping Beauty. https://sleeping-beauty.kontrollfeld.com.
[15] The IoT-LAB Testbed. https://www.iot-lab.info/hardware/.
[16] The RIOT Summit (Annual Conference). http://summit.riot-os.org.
[17] Unwired Devices. https://www.unwireddevices.com/products/.
[18] Zephyr Project. https://www.zephyrproject.org.
[19] C. Adjih et al. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *Proc. of IEEE World Forum on IoT*, December 2015.
[20] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A Survey of Information-Centric Networking. *IEEE Communications Magazine*, 50(7):26–36, July 2012.
[21] M. Antonakakis et al. Understanding the Mirai Botnet. In *Proc. of 26th USENIX Security Symposium*, 2017.
[22] E. Baccelli, J. Doerr, S. Kikuchi, F. Acosta, K. Schleiser, and I. Thomas. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom*, March 2018 (to appear).
[23] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, 2013. IEEE Press.
[24] R. Barry. FreeRTOS embedded operating system. http://www.freertos.org.
[25] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, IETF, May 2014.
[26] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. 2001.
[27] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN*, pages 455–462. IEEE Computer Society, 2004.
[28] P. Gaur and M. P. Tahiliani. Operating Systems for IoT Devices: A Critical Survey. In *2015 IEEE Region 10 Symposium*, pages 33–36, May 2015.
[29] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, 2015.
[30] O. Hahm et al. Low-power Internet of Things with NDN and Cooperative Caching. In *Proc. of 4th ACM Conference on Information-Centric Networking (ICN)*, 2017.
[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.
[32] Hochschule Offenburg. *Documentation of the emb6 Network Stack*, v0.1.0 edition, 2015. https://github.com/hso-esk/emb6/blob/b4ec037cd38c0f87013e3f0fb811f0f6da746f75/doc/pdf/emb6.pdf.
[33] P. Kietzmann, C. Gündogan, T. C. Schmidt, O. Hahm, and M. Wählisch. The Need for a Name to MAC Address Mapping in NDN: Towards Quantifying the Resource Gain. In *Proc. of 4th ACM Conference on Information-Centric Networking (ICN)*, pages 36–42, New York, NY, USA, September 2017. ACM.
[34] D. Kreutz et al. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, 2015.
[35] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch. Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things. Technical Report arXiv:1801.02833, Open Archive: arXiv.org, January 2018.
[36] P. Levis. Experiences from a Decade of TinyOS Development. In *Proc. of OSDI*, pages 207–220, Berkeley, CA, USA, 2012. USENIX Association.
[37] L. Mirani. Chip-makers are Betting that Moore's Law Won't Matter in the Internet of Things. *Quartz*, 2014.
[38] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. McCann, and K. K. Leung. A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities. *Wireless Communications, IEEE*, 20(6):91–98, 2013.
[39] M. M. Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.
[40] H. Will, K. Schleiser, and J. H. Schiller. A Real-time Kernel for Wireless Sensor Networks Employed in Rescue Scenarios. In *IEEE LCN*, 2009.