Choosing the right RTOS for IoT platform

Aleksandar Milinković
Belgrade University, School of Electrical Engineering
Belgrade, Serbia
amilinko@gmail.com

Stevan Milinković
Union University, School of Computing
Belgrade, Serbia
smilinkovic@raf.edu.rs

Ljubomir Lazić
Metropolitan University, Faculty of Information Technology
Belgrade, Serbia
ljubomir.lazic@metropolitan.ac.rs

Abstract – **To fully take advantage of the opportunity offered by the Internet of Things (IoT), manufacturers of embedded systems must meet multiple challenges, which cannot be addressed without a real time operating system. In this paper we propose some criteria that are important for the selection of such an operating system, which could be used for different classes of IoT platforms. We give a short survey of open-source solutions that are applicable in the IoT systems based both on ARM Cortex-M and TI MSP430 microcontrollers. Selected operating system was ported to the target test platform, but it turned out that porting was not as trivial as it seemed at first glance. The proposed platform was tested in a wireless environment with an intention to be applied in home automation.**

Key words – RTOS; constrained devices; Internet of Things;

I. INTRODUCTION

Internet of Things (IoT) provides communication between different objects integrated into a global network as well as the people and these objects. In order to achieve this, it is necessary to ensure that the cost of such facilities is low, and a software that enables their operation should be adapted to the environment in terms of performance, memory size and power consumption. Today, the IoT can be associated with all that is called "smart", e.g. smart cities, smart metering, smart energy, building automation, connected home, smart grid, eHeatlh, etc. Simply put, the IoT is the network of physical objects containing embedded technology that can be accessed through the Internet [1].

The components that are building blocks of IoT have very limited resources, with more restrictions than it is usual for embedded systems. Therefore, the term constrained device [2] was recently introduced in order to define the difference between such a system and desktop computer. This especially applies to the significantly reduced energy, lower computing power, as well as significantly reduced amount of memory. Bearing in mind that the constrained devices are mainly based on microcontrollers, which usually don't have memory management units, their system software exclude solutions that are common in embedded systems. In addition, there is a need to work in real time in some applications, which further narrows our choice. Consequently, real-time operating systems must satisfy performance demands, offer hard real-time response and handle memory constraints, but increasingly they also need to deliver capabilities demanded by the new, highly connected, security conscious, remotely managed world of machine to machine networks and IoT.

In this work our goal was to build IoT platforms with the following characteristics [2]:

First platform:
- Class of Constrained Devices: C2
- Class of Energy Limitation: E9 (no direct quantitative limitations to available energy)
- Strategy of Using Power for Communication: P9 (Always-on)
- ARM Cortex-M microcontroller

Second platform:
- Classes of Constrained Devices: C0 and C1
- Class of Energy Limitation: E1 (Period energy-limited)
- Strategy of Using Power for Communication: P1 (Low-power)
- TI MSP430 microcontroller

A. *Linux solution*

Linux certainly is a robust, developer-friendly OS that has been getting attention as a platform for IoT devices. Linux has matured into a mainstream embedded operating system for many applications. Yet, Linux has a major disadvantage when compared to a real-time operating system: memory footprint. Even though Linux can be trimmed down by removing tools and system services that are not needed in embedded systems, it still remains a large piece of software. It simply will not run on 8 or 16-bit MCUs, and even many newer 32-bit MCUs do not have enough onboard RAM for the Linux kernel. The ARM Cortex-M series is a good example. There are hundreds of different MCUs that are based on the popular Cortex-M architecture, which typically have only a few hundred kilobytes of onboard memory. Linux will never run on these chips.

B. *Industrial or consumer application*

The software requirements for industrial and consumer IoT devices can differ quite a bit. Although they might share a common kernel and low-level services, the middleware required by their applications can be radically different.

Results are part of the research that is supported by Ministry of Education and Science of the Republic of Serbia, Grants No. III-45003 and TR-35026.

An industrial IoT device, such as a wireless sensor node, is a low-power, low-cost device that may run entirely on battery. Such a device might typically use a Cortex-M MCU. It would make use of a highly efficient network protocol such as 6LoWPAN to reduce transmission time and save power. And it would communicate over short distances wirelessly using Bluetooth or low-power Wi-Fi, or else use Ethernet.

On the other side, the software requirements for consumer IoT device are much greater. It might need a Java VM, and may well make use of a vertical market protocol such as AllSeen, Continua Alliance, HomePlug/HomeGrid, or 2net. Such a device might use a Cortex-M or a Cortex-A processor.

C. Scalability

A common engineering solution for networked embedded systems is to use two processors in the device. In this arrangement, an 8 or 16-bit MCU is used for the sensor or actuator, while a 32-bit processor is used for the network interface. That second processor runs an RTOS.

IoT devices will still contain a mixture of small and large MCUs for years to come. A scalable RTOS that runs on a variety of 16 and 32-bit MCUs will allow us to meet tight memory requirements and reduce processor demands.

D. Modularity

The IoT device will require a modular operating system that separates the core kernel from middleware, protocols, and applications. The reasons are ease of development, and keeping the memory footprint of the software to a minimum. Using a modular RTOS simplifies our development process, especially if we are developing a family of devices with different capabilities. Relying on a common core allows the entire family of devices to share a common code base, while each device is customized with only the middleware and protocol stacks required by the application.

This approach also allows for a smaller memory footprint in our device. Unlike a monolithic operating system that bundles an entire suite of software together, a modular RTOS allows us to tailor the embedded software for our device, requiring less RAM and flash memory, and reducing the costs.

E. Connectivity

Network connectivity is essential to the Internet of Things. Whether we are talking about wireless sensor nodes in a factory, or networked medical devices in a hospital, the industry now expects embedded devices to be connected to each other, and to communicate with corporate or public networks. Our RTOS of choice should support communications standards and protocols such as IEEE 802.15.4, Wi-Fi, and Bluetooth. Our device must be able to connect to IP networks using bandwidth-efficient protocols such as 6LoWPAN. It also must be able to use wired media, such as Ethernet.

An RTOS will allow us to select the specific protocol stacks we need, saving memory on the device, and reducing our costs. And it can help us retrofit existing devices with new connectivity options without reworking the core of our embedded software.

The operating system for the IoT must take into account all the constraints of hardware while maintaining a high usability for developers. It must function well on systems with different hardware capabilities and capacities. In addition, it is desirable to have some standard interface (e.g., POSIX), for good portability of applications, for minimizing maintenance, as well as to provide the ability to easily connect to other devices on the Internet. However, the ease of use for developers is crucial. In addition to the C language, the use of other programming languages and libraries is highly desirable, for example C ++ and STL, but they strongly depend on the development toolchain. Not all of them have this capability, and even in those cases where it exists, this task remained a challenge on IoT platforms.

II. A SHORT SURVEY

Operating System (OS) for IoT must be designed purposely, regardless if it is an open-source or proprietary, which means that it must take into account all the constraints that exist in such devices.

Depending on their primary purpose, operating systems have different characteristics, but one of the main things which is necessary to take into account is the kernel structure. Kernel implementation may also have historical reasons, but in embedded systems they are of minor importance. Therefore, designers have monolithic, layered and microkernel architectures at their disposal.

Another important aspect in choosing or designing the operating system is scheduler. Scheduling strategy directly affects the system's ability to operate in real time, as well as to support different priorities and ways of interacting with the user. It also has a significant impact on energy consumption of the entire device. Some operating systems provide several different scheduling algorithms and policies.

Finally, the third aspect in the design is the programming model. On some operating systems, all tasks are executed in the same context and without partitioning of the memory address space. Other systems support multi-threading, where each task executes within its own thread and has its own stack.

The programming model is closely related to selected programming language, because it can affect the implementation of the operating system itself, and also determine which programming language will be used by developers when working with this particular operating system.

A. Contiki and Tiny OS

Contiki and Tiny OS are not real time systems, but their presence on the market is dominant (mainly in wireless sensor networks) and therefore they are used for reference. The same reason applies to Linux.

Contiki operating system has a layered architecture, while Tiny OS is built upon a monolithic kernel, as is the case with Linux. The Contiki system is event driven and is similar to that of TinyOS, which uses a FIFO strategy. Linux on the other hand, uses a scheduler that guarantees fair schedule for all tasks, even allowing preemption by timer. Only Linux with real-time extensions has a scheduler for work in real-time.

Programming models with Contiki and TinyOS are defined by events in a way that all tasks are executed in the same context, although they offer a partial multithreading support. Contiki uses a programming language similar to C, but can't use certain keywords. TinyOS is written in the language called nesC, which is similar but not compatible with the C language. Linux, on the other hand, supports true multithreading, it is written in standard C, and it offers support for various programming languages. Compared to Linux, TinyOS and Contiki do not possess several functionalities that would be a great relief to developers, such as programming in standard C and C ++, the standard multithreading, as well as support for real-time (see Table 1) [3].

TABLE 1. KEY CHARACTERISTICS OF TINYOS, CONTIKI, RIOT, AND LINUX. (✓) FULL SUPPORT, (○) PARTIAL SUPPORT, (X) NO SUPPORT.

OS	min RAM	min ROM	C support	C++ support
TinyOS	< 1kB	< 4kB	x	x
Contiki	< 2kB	< 30kB	○	x
RIOT	~ 1.5kB	~ 5kB	✓	✓
Linux	~ 1MB	~ 1MB	✓	✓

OS	multi-threading	MCU w/o MMU	modularity	real-time
Tiny OS	○	✓	x	x
Contiki	○	✓	○	○
RIOT	✓	✓	✓	✓
Linux	✓	x	○	○

TinyOS version 2.1 introduces TOSThreads, fully preemptable user-level application threads library [4], in which programmer can use C and nesC APIs, but outside of that still has to use only nesC.

A TinyOS has to be present in the form of source code or as a library during the compilation of user programs (static linking), providing a common binary program which is then programmed into device. This approach simplifies some things, such as better resource usage analysis, or more efficient optimization. On the other hand, changes in customer applications require a redistribution of the entire operating system. Unlike TinyOS, Contiki has the ability to load individual applications or services during the execution of the operating system on the device, which resembles the mechanisms on general purpose computers.

On top of Contiki basic event-driven kernel other execution models can be used. Instead of the simple event handlers processes can use Protothreads [5]. Protothreads are simple forms of normal threads in a multi-threaded environment. Protothreads are stackless so they save their state information in the private memory of the process. Like the event handler Protothreads can not be preempted and run until the it puts itself into a waiting state until it is scheduled again.

Along with event-driven kernel, Contiki also contains a preemptive multithreading library. It is statically linked with application program only if the program explicitly calls some of its functions. However, each thread from this library must have its own stack, which is not the case in Protothreads [6].

B. FreeRTOS

FreeRTOS is a real-time kernel/scheduler designed to be small enough to run on a microcontroller. Typically a kernel binary image will be in the region of 4kB to 9kB [7].

The scheduler in FreeRTOS has two operating modes: Preemptive and Cooperative. Cooperative scheduling avoids the reentrance problems faced by preemptive scheduling. This is because tasks that are being executed can only be interrupted at positions permitted by developer and not arbitrarily. It is important that although real-time performance is affected at task levels, interrupts continue to enjoy real-time responses by using semaphores. Tasks with highest priority are executed first, and for more than one high priority tasks, round-robin mechanism is used.

In FreeRTOS, when any task is created, the kernel does two memory allocations. The time required for memory allocation in the Task Control Block is fixed. The time required for the initialization of task stack is proportional to the complexity of the task, i.e. the stack size required.

FreeRTOS has three models for allocation of memory spaces. The simplest models allocate a fixed memory to each task but do not deallocate it. Thus memory cannot be reused and leads to the wastage of memory space. The second model allows allocation and de allocation and uses a best-fit algorithm to find free space in memory. The most complex model uses custom algorithms for specific requirements of task.

The greatest disadvantage of freeRTOS is that it doesn't implement advanced mechanism for shared resources like priority ceilings to avoid priority inversion.

C. FreeRTOS+Nabto

FreeRTOS doesn't have HAL/device drivers, and that is the way which ensures it's great portability. However, the main consequence for IoT application is that FreeRTOS doesn't provide any comunication at all. Fortunately, by combining the FreeRTOS with Nabto [8] peer-to-peer remote access communication platform, we can harness a simple and secure HTML5 or native application interface for the end users, along with an adaptive and flexible data-acquisition interface for data collection, central analysis and monitoring systems. With Nabto, the IoT device only handles a small amount of data – outgoing data from its sensors and incoming commands. The remainder of any web pages seen by the user is served by a remote server in the cloud. Page seen by the user are assembled from the two sources in the PC or phone by a Nabto plug-in in its web browser. In use, the plug-in goes first to the cloud server. The server knows where both items are and serves a specification file to generate a web server interface, mediating direct connection to the device. The external device can work behind firewall, doesn't need a file system, and doesn't need TCP/IP stack. For development, and for use across a local, no cloud server is needed. Instead the web content is cached on the user's PC or smart phone. If local, the plug-in will just search and come up with a list of devices to connect to. FreeRTOS+Nabto enabled devices can also be accessed over the local network in the absence of Internet connectivity.

D. ChibiOS/RT

ChibiOS/RT [9] is designed for deeply embedded real time applications where execution efficiency and compact code are important requirements. This RTOS is characterized by its high portability, compact size and, mainly, by its architecture optimized for extremely efficient context switching. While the ChibiOS/RT kernel can be used even alone, the RTOS also offers other subsystems, e.g. an HAL abstracting many common device drivers and integration with other open source projects like file systems, networking stacks etc.

Everything in the kernel is static, i.e. nowhere memory is allocated or freed. Actually, there are three allocator subsystems but those are optional and not part of core OS. Dynamic services are built as a layer on top of the fully static kernel. The kernel has no internal tables, and there is nothing that must be configured at compile time or that can overflow at run time. There are no upper bounds, the internal structures are all dynamic even if all the objects are statically allocated.

System APIs have no error conditions, all the previous points are finalized to this objective. Everything we can invoke in the kernel is designed to not fail unless we pass bad parameters, (e.g. stray pointers). The APIs are not slowed down by parameter checks. Parameter and consistency checks do exist, but only when the related debug switches are activated. All the static core APIs always succeed if correct parameters are passed. Each API function should have the parameters we would expect for that function and do just one thing, with no options.

The OS code is fast in the first place, that is, the focus is on speed and execution efficiency and then on code size. This does not mean that the OS is large. The kernel size with all the subsystems activated weighs around 5.5kB for STM32 Cortex-M3 microcontroller.

E. RT-Thread

RT-Thread is an open source real-time operating system for embedded devices [10]. The kernel has real-time multi-tasking scheduler, semaphores, mutexes, mail boxes, message queues etc. In addition, this RTOS has support for various device drivers and component-based services, e.g. a shell, virtual file system (FAT, YAFFS, UFFS, ROM/RAM file system etc), TCP/IP protocol stack (lwIP), POSIX interface, etc. It is possible for user to add new components. The kernel itself occupies as low as 3kB in ROM and 1kB in RAM.

Real-time threads within the operating system use an object-oriented design approach, and the built-in kernel object management system that can access / manage all kernel objects. Kernel objects contain most of the kernel facilities, and can be static or dynamic objects. Therefore, system does not depend on the specific memory allocation, and scalability has been greatly enhanced.

Priority-based scheduling algorithm is fully preemptive, supporting up to 256 thread priorities. The same priority threads can be scheduled in round-robin fashion. Scheduler finds the next thread for execution in constant time (O (1)). The system does not limit the number of threads. It depends only on memory associated with a specific physical platform.

RT-Thread system supports semaphores, mutexes and other interthread synchronization mechanism. Mutexes use priority inheritance mechanisms. System also has an event, mailbox, and message queue communication mechanisms. Multi-event (“or trigger”, “and trigger”) is supported when thread waits for multiple events. Mailboxes are of fixed length (4 bytes), which provide higher efficiency than message queue. System supports static and dynamic heap memory pool management. Kernel memory manager uses Buddy and/or Slab allocator.

Our main problem with RT-Thread is the documentation, which is written in Chinese language.

F. Erika Enterprise

Erika Enterprise [11] is a free of charge, open-source RTOS implementation of the ISO 17356 API (derived from the OSEK/VDX API), and it is OSEK/VDX certified. The kernel is organized in a modular fashion and it is fully configurable both in terms of services and kernel objects (tasks, resources, and events). It allows the user to include only those services strictly required by the application, thus achieving a minimal memory footprint of 2kB, up to more complete configurations. It is available for a wide variety of 8, 16, and 32 bit MCUs (including multicores) and supports advanced scheduling mechanisms. The kernel modular design allows reusing the software modules in different applications, speeding up the development of new projects or the upgrade of existing projects to more powerful architectures, and simplifying the maintenance.

The kernel consists of two layers: the Hardware Abstraction Layer (HAL) and the Kernel Layer. The HAL represents the very low level kernel layer; therefore, different HALs are required for different processors (notice that the Kernel Layer does not change when the Erika system is ported on different platforms) The HAL contains the hardware dependent code to manage the context switches and to handle the interrupt requests. The Kernel Layer is composed by a set of modules for task management and real-time scheduling policies. Fixed priority with preemption threshold and Earliest Deadline First (EDF) with preemption threshold are currently supported by the kernel. Both use the Stack Resource Policy (SRP) to share resources between threads and to share the system stack among the threads while preserving time predictability.

Erika supports stack sharing to reduce RAM usage, which is very important for systems with very little RAM. That is, there is no need for a separate stack for all the tasks. On the other side, it does not have some of the primitives, like message passing (although some prototypes are there since longtime, but were never integrated in the main build). Erika has support for Scilab/Scicos code generation, and also supports ZigBee protocols using openZB [12].

G. RIOT

RIOT (Real-time operating system for IoT) fills the gap between operating systems of wireless sensor networks and traditional operating systems. In addition, this operating system is designed to take care about energy efficiency of the device, to occupy as less memory space as possible, and to have a unique API, regardless of the underlying hardware.

RIOT is based on a microkernel architecture, which is inherited from FireKernel [13], thereby supporting multithreading using a standard API. Packed with features inherited from FireKernel, RIOT also provides support for C++, enabling the use of powerful libraries, such as Wiselib [14]. As standard, it includes support for TCP / IP stack network. This modular approach makes RIOT robust against failures of its individual components, providing high reliability with a programmer-friendly API.

RIOT allows programmers to create as many threads as they need. The only constraint is the amount of available memory and the size of the stack for each thread. Thanks to the kernel message API and using these threads, it is possible to implement distributed systems in a simple way.

To fulfill strong real-time requirements RIOT enforces constant periods for kernel tasks (e.g., scheduler run, inter-process communication, timer operations). An important prerequisite for guaranteed runtimes of O(1) is the exclusive use of static memory allocation in the kernel. Yet, dynamic memory management is provided for applications. Constant runtime of the scheduler is achieved by using a fixed-sized circular linked list of threads.

Low complexity of kernel functions is a main factor for the energy efficiency of RIOT. The duration and occurrence of context switching is minimized. In RIOT context switching is performed in two cases: (1) a corresponding kernel operation itself is called, e.g. a mutex locking or creation of a new thread, or (2) an interrupt causes a thread switch. Fortunately, the first case will occur rarely, since in majority of applications every thread is created once. On the other hand, when RIOT's kernel gets called out of an interrupt service routine, saving the old thread's context is not required and thus a task switch can be performed in very few clock cycles.

III. CLASS C2 IOT PLATFORM IMPLEMENTATION

As we have seen, each RTOS is very good in its particular domain, but taking into consideration our requirements, we have chosen RIOT. On the high end in terms of hardware MCU and memory capacities, RIOT competes mainly with Linux. Compared to Linux, RIOT can scale down to orders of magnitude less memory requirements and supports built-in energy efficiency and real-time capabilities. On the low end in terms of hardware MCU/memory capacities, RIOT competes mainly with Contiki, TinyOS, and FreeRTOS (see Table 1). Compared to Contiki and TinyOS, RIOT offers real-time capabilities and multi-threading. In contrast to FreeRTOS, RIOT provides native energy efficiency and a full-featured OS including up-to-date, free, open-source interoperable network stacks (e.g., 6LoWPAN), instead of just a kernel. RIOT also offers standard POSIX APIs and the ability to code in standard programming languages (C and C++) using standard debugging tools, thus reduces the learning curve of developers and the software development lifecycle process.

Default protocol integration in RIOT is mainly driven by latest IETF/IRTF activities [15]. Currently, RIOT supports basic networking protocols including 6LoWPAN, RPL, IPv6, TCP, UDP, CoAP, and provides CCNlite to experiment with content-centric networking.

The C++ capabilities of RIOT enable powerful libraries such as the Wiselib, which includes algorithms for routing, clustering, timesync, localization, and security.

There are a number of other topics in current development, such as dynamic linking support, Python interpreter, CBOR (an alternative for JSON), energy profiler, etc.

RIOT currently supports the number of microcontroller families: ARM7, Cortex-M, AVR/ATmega and MPS430, on different platforms, such as Intel Galileo, Betty, TelosB, STM Discovery, Arduino Mega 2560, etc.

In addition, to overcome the issue of specialized hardware availability, RIOT can also be run as a native process on Linux and MacOS. This facilitates development because such a native process can be analyzed using readily available tools (e.g., gdb, Valgrind). A RIOT process is accessible via shell, the UART interface, or the virtual link layer interface (TAP).

At the moment, we describe our results only for the first platform (class C2 - see section I). As a proof of concept, our target hardware for RIOT operating system is the development board EasyMx PRO v7 for STM32 ARM manufactured by Mikroelektronika [16]. It contains many on-board modules necessary for development variety of applications, including multimedia, Ethernet, USB, CAN and other, as well as mikroProg programmer and debugger. Board is delivered with MCU card containing STM32F407VGT6. It is in many aspects very similar to the already supported STM32F4discovery.

However, porting of the original source code for M4 platform (*RIOT/cpu/stm32f4/*) was not so straightforward as it should be. The main reason was the MikroC compiler. Directory *RIOT/cpu/arm_common/* contains hardware specific, yet common routines for Cortex-M processors. All of them are written in assembly language, however porting them to MikroC compiler was not trivial task. MikroC does not allow separate files containing only assembler functions. Instead, assembly language routines should be embedded in some C file, and within C function, which often unnecessary generates a stack frame. Their assembler is quite restrictive in terms of syntax, and it is documented only with a few sentences in Help file. For example, interrupt handler should be wrapped with C function, which in its declaration tells compiler where is the vector in interrupt vector table for that particular interrupt.

In assembly code callable from C, care should be taken of compiler optimization level in order to avoid different methods of parameter passing (stack or registers). The same applies when using microprocessor registers within assembly code.

Other specific code we had to write were device drivers for radio modules. Two different radio boards were used: 802.15.4 on all boards, plus 802.11 on data sink board.

Mikroelektronika BEE Click is an accessory board featuring 2.4 GHz IEEE 802.15.4 radio transceiver module MRF24J40MA (Microchip). This module includes an integrated PCB antenna and matching circuitry and is connected to the microcontroller via a SPI interface. Virtual functions from RIOT radio interface driver are implemented directly by using routines given in BEE click example for ARM Cortex-M [17].

WiFi PLUS Click features Microchip MRF24WB0MA, IEEE 802.11 compliant module as well as MCW1001 (Microchip) companion controller with on-board TCP/IP stack and appropriate 802.11 connection manager. WiFi PLUS click communicates with target board via UART interface. Due to highly modular design of the operating system, it was much easier to write device drivers for these radio modules.

During our preliminary test, eight nodes (including the sink) were in the same room, all within radio range with each other, resembling the full mesh network. Nodes are programmed as servers using socket API (part of network related POSIX wrapper of RIOT), i.e. the application threads are blocked in *accept*, until the request for connection is received. Only one connection is possible at the time. There is no payload in incoming packet. Immediately upon establishing connection, the server reads integer value from ADC channel 3 and sends it back to the sink node. The sink node application is programmed as client, i.e. it polls all nodes by executing instruction *connect* to the particular address/port and waiting for the response. After the message is received from the RIOT node, it is forwarded via WiFi PLUS Click module toward the desktop computer.

IV. CONCLUSION

We have made analysis of some existing solutions and chose an open-source operating system which we ported and compiled in environment that was not originally written for. Namely, original RIOT is written by means of the GCC toolchain, which has a very little in common with our development environment, including the C compiler itself. On the other side, unique C language development tools from Mikroelektronika are made to support broad range of very different platforms, such as ARM STM32, PIC, PIC32, dSPIC, AVR, Tiva and even 8051 series of microcontrollers on number of development boards. That is much larger hardware base than originally provided by RIOT based on the GCC toolchain and appropriate drivers.

The idea behind our work is to build a common hardware/software IoT platform that can allow us to make smart objects by just connecting them and building an application for them. The chosen operating system for our platform (RIOT) is a real-time multi-threading operating system aiming to ease development across a wide range of IoT devices. Designed for energy-efficiency, reliability, real-time capabilities, small memory footprint, modularity, and uniform API access, RIOT provides several libraries such as Wiselib, as well as a full IPv6 stack for connecting constrained systems to the Internet. Unfortunately, porting RIOT to Mikroelektronika development board turned out to be not so simple, mainly because it was not possible to use their development tools to build the system in a straight manner. For example, MikroC for ARM uses a proprietary encoded (compressed) library format. Mikroelektronika is secretive and protective of their libraries and compiler inner workings so we cannot use, or make conversion from external object files. On the other hand, their development and add-on boards are very well documented, including detailed schematic diagrams. Therefore, they remain in our attention.

REFERENCES

[1] A. Milinković, S. Milinković, and Lj. Lazić, "Some experiences in building IoT platform", Telfor '14, Belgrade, Serbia, 25-27 Nov. 2014.

[2] C. Bormann, M. Ersue, and A. Keranen, *Terminology for Constrained-Node Networks*, IETF, RFC 7228, May 2014.

[3] O. Hahm, E. Baccelli, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," Proc. of the 32nd IEEE International Conference on Computer Communications (INFOCOM), Poster Session, April 2013.

[4] W. P. McCartney, and N. Sridhar, "Stackless preemptive multi-threading for TinyOS," in 2011 International Conference on Distributed Computing in Sensor Systems and Workshops, Barcelona, Spain, 2011, pp. 1–8,

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in Proc. of the 4th International Conference on Embedded Networked Sensor Systems, Boulder, Colorado, USA, pp. 29–42.

[6] T. Reusing, "Comparison of Operating Systems TinyOS and Contiki," Seminar SN SS2012, Network Architectures and Services, August 2012, pp. 7–13.

[7] FreeRTOS. Availiable: http://www.freertos.org/

[8] Nabto. Available: http://nabto.com/

[9] ChibiOS/RT. Available: http://www.chibios.org/dokuwiki/doku.php

[10] RT-Thread. Available: http://www.rt-thread.org/

[11] Erika Enterprise. Available: http://www.erika-enterprise.com/

[12] opneZB. Available: http://www.open-zb.net/

[13] H. Will, K. Schleiser, and J. H. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios", in Proc. of IEEE Conference on Local Computer Networks (LCN), Zürich, Switzerland, 2009, pp. 834–841.

[14] T. Baumgartner, I. Chatzigiannakis, S. Fekete, C. Koninis, A. Kröller, and A. Pyrgelis, "Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Networks," Wireless Sensor Networks, LNCS, vol. 5970, 2010, pp 162-177.

[15] A. Y. Ding, J. Korhonen, T. Savolainen, M. Kojo, J. Ott, S. Tarkoma, and J. Crowcroft, "Bridging the Gap Between Internet Standardization and Networking Research," SIGCOMM Comput. Commun. Rev., vol. 44, no. 1, pp. 56–62, Jan. 2014.

[16] http://www.mikroe.com/easymx-pro/stm32/

[17] http://www.libstock.com/projects/view/242/bee-click-example/