Running SmartMesh on the MAX32655 with MicroPython

Luiz Sampaio, Kate O'Riordan, Brian Coffey, Lance Doherty, Thomas Watteyne

▶ **To cite this version:**

Luiz Sampaio, Kate O'Riordan, Brian Coffey, Lance Doherty, Thomas Watteyne. Running SmartMesh on the MAX32655 with MicroPython. IECON 2024 - 50th Annual Conference of the IEEE Industrial Electronics Society, Nov 2024, Chicago, IL, United States. hal-04828853

HAL Id: hal-04828853
https://inria.hal.science/hal-04828853v1

Submitted on 10 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Running SmartMesh on the MAX32655 with MicroPython

Luiz Sampaio, Kate O'Riordan, Dara O'Sullivan, Brian Coffey, Lance Doherty, Thomas Watteyne
Analog Devices
first.last@analog.com

***Abstract*—Industrial low-power wireless networks exhibit unique requirements in terms of reliability, battery lifetime and security. Time Synchronized Channel Hopping is a networking technique created to address these needs, which was standardized by working group IETF 6TiSCH. Analog Devices' SmartMesh product lines have been the best-in-class TSCH implementation, exhibiting over 99.999% wire-like end-to-end reliability, a decade of battery lifetime, and certified security. With over 100,000 networks deployed, SmartMesh plays a market-leading role. Today, SmartMesh runs on the LTC5800, and often requires customers to drive it from a second external micro-controller, which increases cost. This paper introduces a port of SmartMesh to the MAX32655, a dual-core microcontroller and radio System on Chip: the RISC-V core runs the communication stack, the ARM Cortex-M4 core is left available to the customer. We show how that ARM Cortex-M4 core can run a MicroPython interpreter for faster prototyping and time-to-market. A simple script, designed to transmit packets over a SmartMesh network using MicroPython, requires only an additional 6 kB of RAM. Furthermore, by compiling this script into byte code, the processing time can be reduced to a quarter of its original duration.**

***Index Terms*—Industrial IoT, TSCH, SmartMesh, MAX32655, MicroPython**

I. Introduction

Industrial Wireless Networks (IWNs) have become a cornerstone of Industry 4.0, permeating the industrial landscape [1]. These networks allow communication between different parts of a factory, thereby enabling predictive maintenance, intelligent motion control, and robotics. However, it is important to note that general Wireless Sensor Networks (WSNs) often overlook the specific requirements of the industrial environment, such as reliability, latency, and low power operation. In this context, the core technique of IEEE802.15.4 [2], Time-Slotted Channel Hopping (TSCH), emerges as a significant development. TSCH, a medium access control (MAC) sublayer, enables time and frequency division multiplexing. This technique is particularly effective for ultra-low-power applications in environments characterized by high interference and multi-path fading.

To enhance the reliability of a TSCH network, Analog Devices has created its SmartMesh IP[1] product line. These products feature over 99.999% data reliability in challenging RF environments, while also ensuring years of battery life across an entire wireless mesh network. A SmartMesh network is composed of motes wirelessly connected to a manager over a self-forming multi-hop mesh network, as shown in Fig. 2.

[1] https://www.analog.com/en/solutions/smartmesh.html

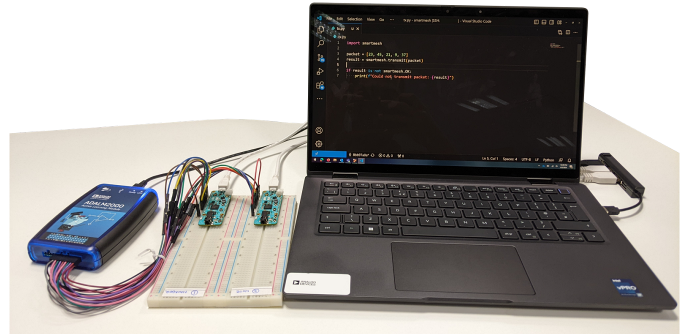

Fig. 1. Two MAX32655FTHR boards running SmartMesh with MicroPython connected to a PC

Today, SmartMesh runs on the LTC5800 System-on-Chip (SoC), which integrates a low-power radio and an ARM Cortex-M3 micro-controller. This micro-controller runs the SmartMesh protocol stack and allows an external micro-controller to drive it over a serial interface. The LTC5800 implements the MAC layer associated with TSCH in hardware, positioning it as the industry's most reliable and lowest power standards-based low-power wireless product. To drive the LTC5800, a customer typically has to drive it from a second external micro-controller, which contributes to the overall cost of their solution.

The MAX32655 addresses this as it feature two processing cores: a RISC-V core which can run the SmartMesh networking stack and drive the low-power wireless radio, and a ARM Cortex-M4 which is fully available to the (customer) application. The development kit of the MAX32655 is built around the popular "feather" form factor, as shown in Fig. 1. No additional equipment, other than a USB cable, is needed to program it. Porting SmartMesh to the MAX32655 offers a more integrated user experience and removes the need for a second external micro-controller, giving more memory to an application running in the same chip and providing a better set of peripherals when compared to the LTC5800.

The contributions of this paper are threefold:

- We port SmartMesh to the MAX32655. This results in a perfectly integrated user experience for achieving wire-like reliability on wireless sensor networks.
- We run MicroPython on the ARM Cortex-M4 core of the

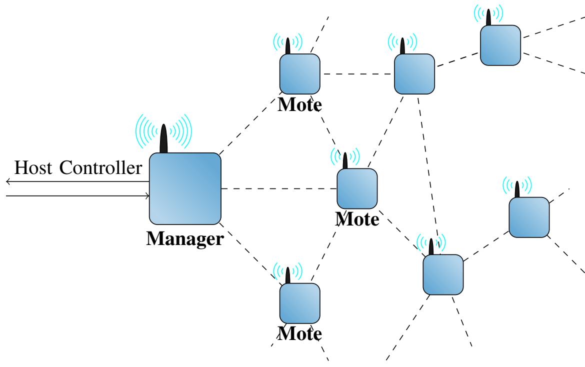

Fig. 2. SmartMesh IP Network

MAX32655. This makes prototyping considerably faster, significantly lowering time-to-market.
- We conducted a measurement campaign to evaluate the performance of this port. The processing time required to transmit a packet and the memory footprint were measured, utilizing both a pure-C and MicroPython interface to the stack. Our findings indicate that a mere additional 6 kB of RAM is sufficient to run a simple script for transmitting packets over SmartMesh. Furthermore, by compiling the script into byte code, we were able to reduce the processing time to a quarter of its original value.

The remainder of this paper is organized as follows. Section II presents the time-slotted channel hopping technology as implemented by the SmartMesh IP products. Section III describes the porting process to the MAX32655, with a highlight on the SmartMesh software architecture. Section IV introduces MicroPython and provides details on its integration with the SmartMesh network stack. Section V presents the performance evaluation of SmartMesh on the MAX32655 with MicroPython. Finally, Section VI concludes this paper and presents avenues for future work.

II. Crash Course on Time-Slotted Channel Hopping

Time-slotted Channel Hopping (TSCH) was incorporated in the 2015 revision of the IEEE802.15.4 standard. This standard targets embedded devices used in industrial applications where energy consumption, reliability, and cost are critical requirements. In a TSCH network, nodes are tightly synchronized. This synchronization allows nodes to know in advance when to activate or deactivate their radios for transmission and reception. To mitigate multi-path fading and external interference, nodes also employ channel hopping. Therefore, TSCH uses both time- and frequency-division multiplexing. This approach allows nodes to use different channels in subsequent communications [3].

Time is divided into timeslots. During a timeslot, a node knows exactly when to activate or deactivate its radio. Fig. 3 presents an IEEE802.15.4 TSCH timeslot timing template for both transmitting and receiving nodes. The packet transmission

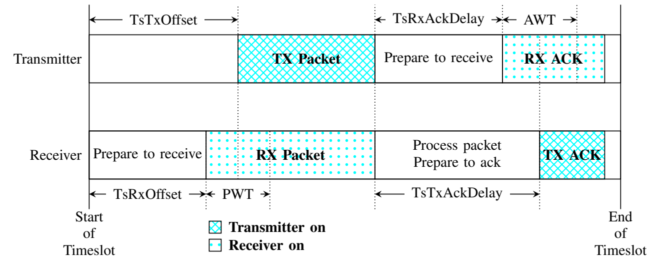

Fig. 3. IEEE802.15.4 TSCH Timeslot Template

occurs exactly at `TsTxOffset`. Once the packet transmission is complete, the transmitter waits for `TsRxAckDelay`, then activates its radio and listens for an acknowledgement packet (ACK) for `AWT`. Similarly, the receiver waits for `TsRxOffset` from the beginning of the timeslot and activates its radio, waiting for a packet for `PWT`. After successfully receiving the packet, it waits for `TsTxAckDelay` and transmits an ACK packet. Within a timeslot, the synchronization and the timeslot template ensure that the motes only activate their radios when they expect to receive a packet.

Timeslots are organized into slotframes that can be visualized as a matrix-like schedule. This schedule determines whether a timeslot is active (scheduled) or not. If active, it specifies the type of timeslot (transmit, listen, broadcast) and the frequency at which it should operate. This schedule represents potential communications between nodes, and is managed by a scheduling algorithm. Each scheduled timeslot is represented by a slot offset, which is the slot index from the beginning of the slotframe, and a channel offset, which maps the timeslot to a specific frequency. An Absolute Slot Number (ASN) indicates the number of timeslots since the network was initiated. Fig. 4 shows an example slotframe schedule with 7 timeslots and 4 channel offsets. In this simple example, timeslots are scheduled for advertisement, transmission, and reception. To determine the actual physical channel for transmission, TSCH standards define a channel hopping sequence, referred to as *macHopSeq*. The physical channel for each slot is determined by (1), where *macHopSeqLen* is the number of elements in the *macHopSeq* list and % is the modulo operator.

$$\begin{aligned} \text{COUNTER} &= \text{ASN} + \text{Channel Offset} \\ \text{CH} &= macHopSeq[\text{COUNTER} \,\%\, macHopSeqLen]. \end{aligned} \quad (1)$$

Using (1) ensures that, at each slotframe, a timeslot in a given slot offset operates at a different frequency given that the slotframe length and *macHopSeqLen* are mutually prime.

Several standardization bodies develop standard specifications for IoT devices. The IETF IPv6 over the TSCH mode of IEEE802.15.4e (6TiSCH) Working Group has been actively working on the standardization process of TSCH and on the IP layer adaptation in order to bring IPv6 to industrial low-power wireless networks [4]. Even though the IEEE802.15.4 standard ensures data integrity and confidentiality at the MAC layer, higher layers also employ mechanisms to ensure authentication. The IETF Lightweight Authenticated Key Exchange

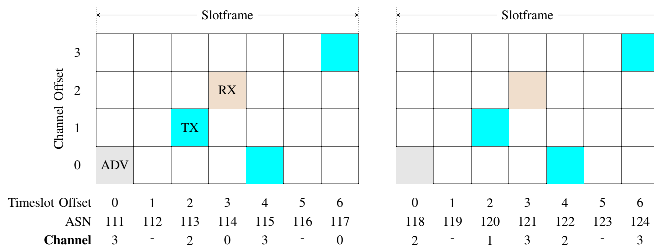

Fig. 4. IEEE802.15.4 TSCH Slotframes

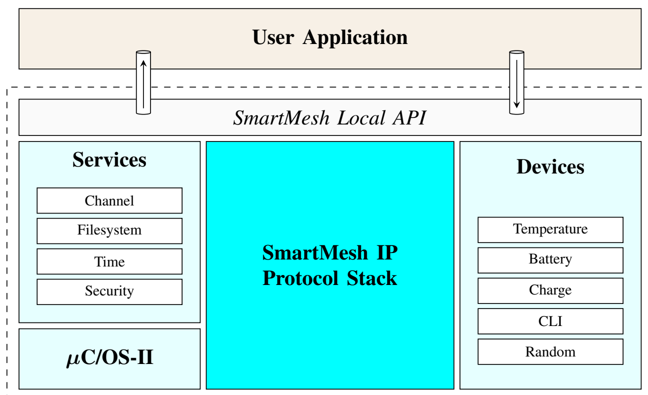

Fig. 5. SmartMesh software block diagram

(LAKE) working group has been working on the standardization of a key exchange protocol for constrained network devices [5].

SmartMesh running on the LTC5800 chip has been deployed in over 100,000 networks, in applications as diverse as refinery process control, energy monitoring and smart cities [6]. TSCH has also been a catalyst for research, with 6TiSCH now being supported by all major open-source IoT implementation, including Contiki-NG, RIOT and OpenWSN [7].

III. Porting SmartMesh to the MAX32655

The MAX32655[2] is an advanced ultra-low-power microcontroller with a 2.4 GHz radio. It features an ARM Cortex-M4F core, a RISC-V core, 512 kB of Flash, 128 kB of SRAM. Additionally, it offers several low power modes and a variety of peripherals, including UART, I2C, SPI, I2S, 1-Wire, ADCs, TRNG, AES. The radio supports BLE 1M, 2M, and Coded versions (S=2 and S=8).

Porting SmartMesh to the MAX32655 enables a perfectly streamlined user experience, without the need of an external micro-controller. Developers can utilize the SDK of the MAX32655 to interface with sensors in a perfectly straightforward manner.

Fig. 5 is a high-level software block diagram of SmartMesh. Customers typically develop an application which interfaces with sensors and actuators connected to the chip, and use SmartMesh to interact wirelessly with the SmartMesh manager. The application interacts with the network using the

[2] https://www.analog.com/en/products/max32655.html

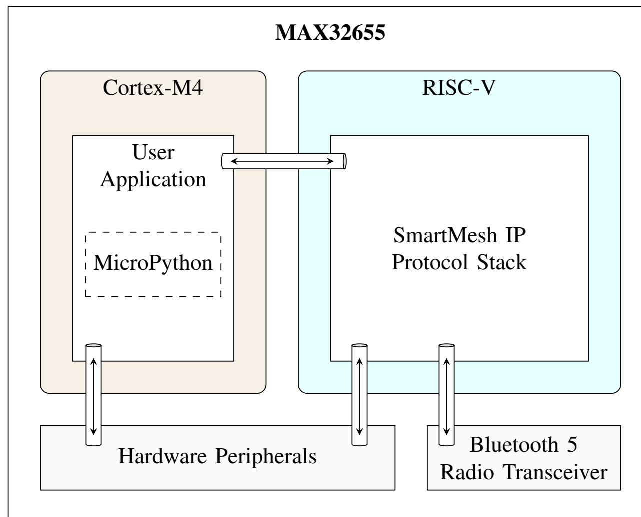

Fig. 6. Block diagram of SmartMesh running on the MAX32655

SmartMesh local API, facilitated by the On-Chip Software Development Kit (OCSDK[3]), which makes SmartMesh services accessible to the user. *Channels* are a software concept at the heart of SmartMesh and serve to transmit and receive data to and from the stack, providing a very clean abstraction between the memory regions of different code parts via the Memory Protection Unit (MPU). SmartMesh also offers a *file system* for the application to create persistent configuration files in Flash memory, a *time* service to alert the user at specific timestamps, and a *security* service for accessing the AES engine. Internally, SmartMesh relies on several devices, such as a temperature sensor, a battery module for battery voltage probing, and a Command Line Interface (CLI) for user interaction.

SmartMesh is built on top of μC-OS-II, an open-source, preemptive, real-time kernel designed for microcontrollers [8]. This kernel has been ported to more than 40 distinct processor architectures, including ARM Cortex-M4, RISC-V, and x86. It is written in ANSI C to ensure portability, and has been certified for use in a commercial aircraft, meeting the requirements of a safety critical system. When compared to other open-source RTOSes, it exhibits superior performance when comparing task switching latency and time to receive events [9].

The LTC5800, a single-core ARM Cortex-M3 chip, implements certain components of the lower MAC layer in hardware, including time slot and slot frame handling. To adapt this for the MAX32655, these hardware-specific elements were re-engineered in software and integrated within the SmartMesh stack. Additionally, the protocol stack, along with its services and devices, was compiled to operate on the RISC-V. This provides customers with access to a complete Cortex-M4 for their applications, as illustrated in Fig. 6.

The process of porting functions such as automatic network formation, packet transmission and reception, and device con-

[3] https://dustcloud.atlassian.net

figuration, has been underway for 10 months. Thanks to the clear separation between the application, network, MAC, and driver layers, the network and application layers could have been reused to maintain existing functionalities. Meanwhile, the MAC and driver layers have been implemented from scratch with 14,798 new lines of code to date.

IV. MicroPython

MicroPython[4] is an implementation of Python 3, written in C, which is optimized to run on microcontrollers or in constrained environments. It is designed to operate in environments with approximately 256 kB of code space and 16 kB of RAM. While it includes some standard modules found in general-purpose Python 3 implementations, some of these have limited functionality. Importantly, the build is highly configurable, allowing users to include only the modules they need.

MicroPython comprises a compiler that translates Python scripts into bytecode, and a runtime interpreter designed to execute the machine instructions corresponding to this bytecode. This design allows users to either supply a script for execution directly on the board, or pre-compile the script to bytecode using a tool provided by MicroPython, known as `mpy-cross`. For rapid development and testing, users can also use an interactive prompt, the REPL (read-eval-print loop), to execute commands in on-the-fly.

While being a developer-friendly and low-complexity language that is suitable for beginners and facilitates rapid coding, MicroPython falls short in performance when compared to a low-level language like C. This has been demonstrated in previous studies targeting the ESP32 platform. For instance, Ionescu *et al.* [10] show that the computation times for CRC-32 and SHA-256 in C code is faster by three orders of magnitude than the same routines running in MicroPython. This finding is corroborated by another study by Plauska *et al.* [11], which also compares MicroPython with TinyGo and Rust. Furthermore, Dokic *et al.* [12] show that the data propagation speed of neural networks analysis using Arduino and MicroPython yield similar results. One strategy that can be employed to reduce processing time is to pre-compile MicroPython scripts into bytecode. This implies that the script will not need to be compiled during runtime, thereby enhancing performance.

To execute MicroPython on the MAX32655, there are two viable options: the *minimal* port as a baseline, or the *embed* port in accordance with the *embedding* example. The *minimal* port involves the implementation of functions to receive a single character from the command line (`mp_hal_stdin_rx_chr()`) and to transmit strings of a specified length (`mp_hal_stdout_tx_strn()`). This can be achieved by following the implementation found in the SDK. On the other hand, the *embed* port exports a collection of source and header files to be incorporated into an existing C project. We chose this approach as it facilitates the execution

[4] https://micropython.org/

```
...
for (i=0; i < PAYLOAD_LENGTH; i++) {
  pkToSend->locSendTo.payload[i] = 0x10+i;
}

dnErr = dnm_loc_sendtoCmd(
  pkToSend,
  PAYLOAD_LENGTH,
  &rc
);
...
```

Listing 1: Example C code snippet to transmit a packet

```
import smartmesh
packet = list(range(N))
smartmesh.transmit(packet)
```

Listing 2: Example MicroPython code to transmit a packet

of both scripts (via the `mp_embed_exec_str` API) and pre-compiled bytecode (via the `mp_embed_exec_mpy` API).

MicroPython's functionality can be extended through the creation of custom modules and functions, written in C, that can be accessed from Python[5]. Consider, for instance, function `dnm_loc_sendtoCmd()` that is part of the OCSDK and is used to send a packet through the network. The OCSDK *01-join* example has a mote transmit packets once it has joined the network. Listing 1 presents a simplified code snippet where a payload is filled and subsequently transmitted.

We created the *smartmesh* MicroPython module. This module contains a function named *transmit*, which accepts a list as input, as demonstrated in Listing 2, and transmits it through the network. This module is a wrapper around the existing API, as depicted in Listing 3, in which the original `dnm_loc_sendtoCmd` API is used.

This module demonstrates how a user can seamlessly use MicroPython on the MAX32655 platform to expedite development with a SmartMesh network infrastructure. It effectively combines Python's ease of learning and its capabilities for rapid prototyping, thereby offering a streamlined development experience.

[5] https://docs.micropython.org/en/latest/develop/cmodules.html

```
mp_obj_t smartmesh_transmit(mp_obj_t a_obj) {
  mp_obj_list_t *p = MP_OBJ_TO_PTR(a_obj);
  ...
  for (int i = 0; i < p->len; i++) {
    pkToSend->locSendTo.payload[i]
            = mp_obj_get_int(p->items[i]);
  }

  int dnErr = dnm_loc_sendtoCmd(
    pkToSend,
    p->len,
    &rc
  );

  return mp_obj_new_int(dnErr);
}
```

Listing 3: SmartMesh transmit implementation inside module

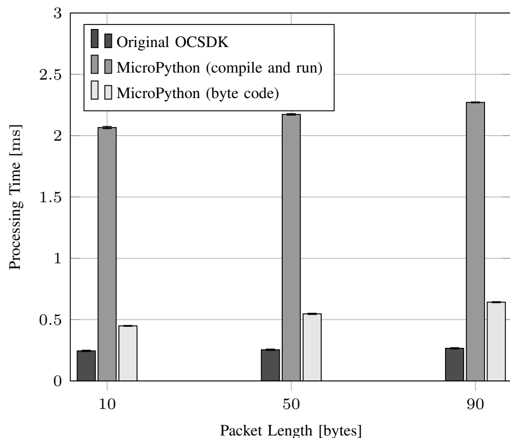

Fig. 7. Processing time to send a packet

V. Performance

In order to showcase the operation of SmartMesh on the MAX32655 and the enhancements brought about by MicroPython, two MAX32655FTHR boards are connected to a PC, as depicted in Fig. 1. These off-the-shelf boards come equipped with a debugger for quick code downloading and testing, and also offer a serial connection for message printing. We connected an ADALM2000 to the boards and used it as a logic analyzer.

Leveraging MicroPython's *embed* port, we create a periodic μC/OS-II task to execute the MicroPython code displayed in Listing 2, both as a script and as bytecode. Upon the task's awakening every second, a GPIO pin is brought high and recorded by the logic analyzer. When the task re-enters sleep mode, it signifies that the packet has been successfully queued for transmission, and the GPIO pin is brought low.

The ADALM2000 is configured to sample the pin at a rate of 100 ksps. We measure the processing time to transmit packets of different lengths, as illustrated in Fig. 7. The Cortex-M4 is set to operate at 100 MHz. As indicated by previous studies, the original OCSDK, implemented directly in C, consistently outperforms other methods. In comparison, the MicroPython script (compiled and run) takes approximately ten times longer to execute. However, it is noteworthy that pre-compiling MicroPython into bytecode reduces the execution time to only 2 to 3 times that of the C implementation. This demonstrates that pre-compiled MicroPython can significantly outperform the standard scripting approach in terms of execution time. In contrast, the user would need to recompile to bytecode each time. A good workflow is hence to use MicroPython scripts for rapid development, followed by only a few compilation cycles.

The code was compiled for execution on the Cortex-M4 using the `arm-none-eabi-gcc` toolchain, which is provided by the SDK, and optimized for size. The `arm-none-eabi-size` command, available in the toolchain, was used to determine Flash and RAM usage, the results of which are presented in Table I. In the context of the experiment, all memory is statically allocated, therefore the `size` command yields a reasonable estimate of RAM usage. As expected, the additional MicroPython code, compiled solely with the simple *smartmesh* module, results in nearly a threefold increase in flash usage. Additionally, RAM usage increases by 6 kB, attributed to the heap allocated for MicroPython. Depending on the specific requirements of the application, users may need to adjust this value.

TABLE I
Flash and RAM memory usage comparison

	Original OCSDK	**MicroPython (compile and run)**	**MicroPython (bytecode)**
Cortex-M4 Flash Usage	58 kB	163 kB	163 kB
Cortex-M4 RAM Usage	18 kB	24 kB	24 kB

VI. Conclusion

This work presents the process of porting SmartMesh to the MAX32655, offering a more modern and cost-effective solution for the Industrial Internet of Things. The study provides an overview of time-slotted channel hopping and its benefits for low-power embedded devices, demonstrating the ongoing standardization work and the commercial products that use it. The advantages of porting SmartMesh to a more modern platform, the MAX32655, are presented, and a MicroPython port is proposed for enhancing user experience. The port's performance is analyzed in terms of the processing time required to send a packet and the memory footprint.

For future work, we are working on a wrapper around the entire OCSDK, allowing the full SmartMesh Local API to be accessed through MicroPython. This enables easier and faster prototyping of applications.

Acknowledgement

This document is issued within the frame and for the purpose of the OpenSwarm project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046. Views and opinions expressed are however those of the author(s) only and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] X. Li, D. Li, J. Wan, A. V. Vasilakos, C.-F. Lai, and S. Wang, "A review of industrial wireless networks in the context of Industry 4.0," *Wireless Networks*, vol. 23, pp. 23–41, 2017.

[2] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer," *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pp. 1–225, 2012.

[3] X. Vilajosana, T. Watteyne, T. Chang, M. Vučinić, S. Duquennoy, and P. Thubert, "IETF 6TiSCH: A Tutorial," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 595–615, 2020.

[4] T. Watteyne, M. R. Palattella, and L. A. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement," Internet Engineering Task Force, Internet-Draft RFC7554, May 2015.

[5] G. Selander, J. P. Mattsson, and F. Palombini, “Ephemeral Diffie-Hellman Over COSE (EDHOC),” Internet Engineering Task Force, Internet-Draft RFC9528, March 2024.
[6] T. Watteyne, L. Doherty, J. Simon, and K. Pister, “Technical Overview of SmartMesh IP,” in *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2013.
[7] T. Watteyne, V. Handziski, X. Vilajosana, S. Duquennoy, O. Hahm, E. Baccelli, and A. Wolisz, “Industrial Wireless IP-based Cyber Physical Systems,” *Proceedings of the IEEE*, vol. PP, no. 99, pp. 1–14, 2016.
[8] J. J. Labrosse, *MicroC/OS-II*, 2nd ed. R & D Books, 1998.
[9] I. Ungurean, “Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers,” *Symmetry*, vol. 12, no. 4, 2020.
[10] V. M. Ionescu and F. M. Enescu, “Investigating the Performance of MicroPython and C on ESP32 and STM32 Microcontrollers,” in *IEEE International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2020.
[11] I. Plauska, A. Liutkevičius, and A. Janavičiūtė, “Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller,” *Electronics*, vol. 12, no. 1, 2023.
[12] K. Dokic, B. Radisic, and M. Cobovic, “MicroPython or Arduino C for ESP32 - Efficiency for Neural Network Edge Devices,” in *Intelligent Computing Systems*, 2020.