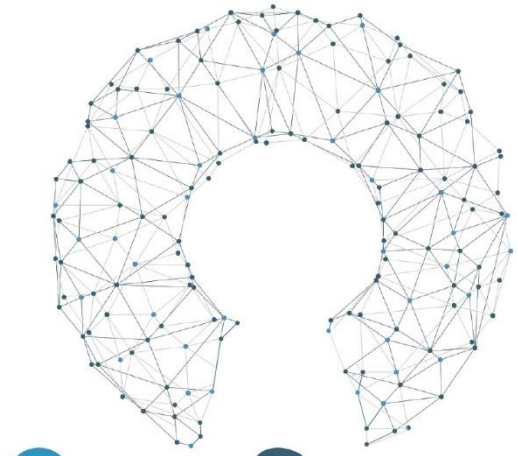

www.openswarm.eu

Call: HORIZON-CL4-2022-DATA-01

Type of action: RIA

Grant agreement: 101093046

Deliverable n°2.1: TSCH Approaches for Network Orchestration and Schedule on General-Purpose Microcontrollers

Work Package n°2: Orchestration of Collaborative Smart Nodes

Task Lead: ADI

WP Lead: ADI

This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046.

Funded by the European Union

Document information	
Author(s)	Luiz Sampaio
Reviewers	Anna Bednarik
Submission date	31-Oct-2024
Due date	31-Oct-2024
Type	Report
Dissemination level	Public

Document history			
Date	Version	Author(s)	Comments
15-Jun-2024	1	Luiz Sampaio	preview
31-Jun-2024	2	Luiz Sampaio	deliverable

Table of Contents

Executive Summary

The focus of this deliverable is the augmentation of Time Synchronized Channel Hopping (TSCH) for the needs of the project. Using SmartMesh, a proven implementation, as one of the wireless communication technologies enables OpenSwarm with highly synchronized wireless nodes, appropriate for applications that require a high level of determinism. The first step is to study the feasibility of porting SmartMesh to general-purpose microcontroller, reducing cost and development time for the testbeds and PoCs. After a thorough study on state-of-the-art microcontrollers with wireless capabilities, we selected the MAX32655, an ultra-low power, dual-core microcontroller, providing a more affordable and streamlined option to develop on top of. The second step is to port SmartMesh to it, which was successfully achieved after developing a new software-defined TSCH on top of the MAX32655's BLE radio. In this context, the network stack will run on its second core (RISC-V), making a full core (ARM Cortex-M4) available for use by the consortium members, simplifying development.

Additionally, as part of the porting process, we developed a new evaluation board tailored to the consortium needs, and it will be distributed to the partners alongside extensive documentation produced to expedite the first steps. Finally, we include some work on the mobility and latency predictability aspects. For mobility, we leverage a strategy used in BLE to advertise on selected channels, accelerating the join process and enabling mobile nodes to rejoin the network quickly. For latency predictability, we present an initial survey on SmartMesh backbone mode and several works in literature optimizing latency and energy consumption, proposing strategies that can be used to achieve a configurable trade-off between both. This work has opened up excited avenue for future work, on both the mobility and latency predictability aspects; we will be exploring those in parallel with remaining OpenSwarm tasks.

Introduction

This deliverable is part of WP2, which aims to develop the core OpenSwarm technologies related to orchestration of collaborative smart nodes, including

networking, security, coordination and management. Specifically, this deliverable contributes to the dependable networking aspect, enabling modifications to TSCH within SmartMesh to meet the needs of the project.

The steps to achieve the goal are the following.

First, study the feasibility of porting SmartMesh to a general-purpose microcontroller thatcan be easily distributed and adopted by the partners. The original SmartMesh runs on the LTC5800, the best-in-class TSCH implementation. However, it has many features implemented in hardware, making it expensive. We studied the MAX326xx (dual-core) family of microcontrollers to determine the most suitable for SmartMesh. The goal is to build SmartMesh on top of the integrated Bluetooth Low-Energy (BLE) Physical Layer. As the microcontrollers under consideration are general-purpose, we carried out extensive viability tests to determine their suitability for SmartMesh, experimenting with clock and timers' capabilities, crystal oscillators accuracy, wake-up times, as well as synchronization between both CPU cores and fine control of the BLE core. Results showed that the MAX32655 is suitable and has been selected as the microcontroller for the project. Beyond the microcontroller, additional hardware will be needed to ensure correct SmartMesh functionality. For example, different architectures of external RTCs and MEMS resonators were considered.

Then, the next step is the actual porting effort of SmartMesh to the chosen microcontroller: MAX32655. This includes the software-defined implementation of the platform-dependent code for TSCH operation. Additionally, the MAX32655 is a dual-core microcontroller, featuring an ARM Cortex-M4 and a RISC-V. In this context, the RISC-V core runs the communication stack, leaving the Cortex-M4 core fully available to the user. To enable communication between cores, an inter-process communication (IPC) library was created, exposing a clear application programming interface (API) to other components in the system. We also upgraded the real-time operating system (RTOS) used by SmartMesh to a more modern, open source, Apache-Licensed version of µC/OS-II. This will make it easier for the partners to integrate into their systems, as now the RTOS source code can be distributed.

Finally, the last step is to modify the communication schedule to support mobility and latency predictability. To support mobility, we follow two approaches: reducing the number of advertising channels and reducing the timeslot length. Theoretically, these changes can accelerate the join procedure by a factor of five, allowing lost motes in a mobile network to quickly rejoin. In terms of latency predictability, we perform a literature survey to understand the techniques that can be used to understand the trade-offs between latency and energy consumption.

This report is organized as follows. We start by presenting an overview of the state-of-the-art in transceivers and a theoretical background of the time-slotted channel hopping technology as implemented by the SmartMesh products. We then describe the new SmartMesh hardware, including the process of evaluating a new microcontroller. Afterwards, we discuss software, providing details on the port of the hardware-implemented functionalities of original SmartMesh. Finally, we present the ongoing work to include mobility and latency predictability.

The appendices also discuss additional results obtained during the project. The results are not directly related to the deliverable but appear as a consequence of the research process. They were submitted as papers alongside the deliverable results.

Key Performance Indicators (KPIs)

The Key Performance Indicator (KPI) for this deliverable is the target end-to-end reliability for a given deadline. The latency predictability aspect controls the number of cells scheduled, which affects the end-to-end deadlines. Today, SmartMesh achieves 99.999% end-to-end reliability. As we haven't started working on the mobility and latency predictability aspects, the updated target of 99.99% is yet to be validated.

Technology Readiness Level (TRL)

Today's TSCH technology is focused on static devices. The ability to use TSCH in mobile networks is at TRL 2, as we can identify many practical applications that would benefit from it, but no experimental proof or detailed analysis has been done. Verifying the mobility and latency predictability aspects in laboratory environment brings TSCH to

TRL 4, After that, the technology will be ready for the Proof-of-Concept phase of OpenSwarm, finally bringing it to TRL 5.

List of Publications

The following papers were created as part of this deliverable.

Accepted:

- Running SmartMesh on the MAX32655 with MicroPython. Luiz Sampaio, Kate O'Riordan, Dara O'Sullivan, Brian Coffey, Lance Doherty, Thomas Watteyne. Conference of the IEEE Industrial Electronics Society (IECON), Chicago, IL, USA, 3-6 November 2024.

Submitted:

- Running µC/OS-II on Embedded Linux in Production. Luiz Sampaio, Kate O'Riordan, Dara O'Sullivan, Brian Coffey, Thomas Watteyne. ACM International Conference on the Foundations of Software Engineering (FSE), Trondheim, Norway, 23-27 June 2025.

Background

State-of-the-Art in Transceivers

The demand for low power transceivers has increased in the last decades, not only for large base stations with multiple antennas, but also for wireless sensor networks (WSN). The ability to deploy a wireless node that lasts a decade without maintenance is necessary for many applications.

At first, transceivers for WSNs were designed as separate integrated circuits with little constraints on power consumption. One example is the CC2420[1], a 2.4 GHz RF transceiver which consumes approximately 18 mA to transmit or receive and must be driven by an external microcontroller via a serial interface.

With increasing need of low power applications, transceivers like the SI4468[2] provided reduced power consumption of approximately 13 mA, but still driven by an external microcontroller. A simple, more integrated solution was needed.

To fill this demand, the LTC5800[3] was launched, integrating a microcontroller and a radio on the same chip. Additionally, it holds the SmartMesh stack, providing the users with a reliable multi-hop mesh network implementation consuming approximately 5 mA to receive and 10 mA to transmit.

Up to the present time, integration level keeps increasing whilst current consumption keeps decreasing. The MAX326xx is a family of advanced ultra-low-power dual-core microcontrollers with a 2.4 GHz radio integrated in the same 8 mm x 8 mm chip. Additionally, the radio can consume as low as 4 mA of current when transmitting or

1 https://www.ti.com/product/CC2420

2 https://www.silabs.com/wireless/proprietary/ezradiopro-sub-ghz-ics/device.si4468

3 https://www.analog.com/en/products/ltc5800-ipm.html

receiving. This allows a clear separation for products in WSNs, as the network stack can run on a dedicated core, giving freedom to the user to use the other core for the application layer.

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 (Vilajosana 2020).

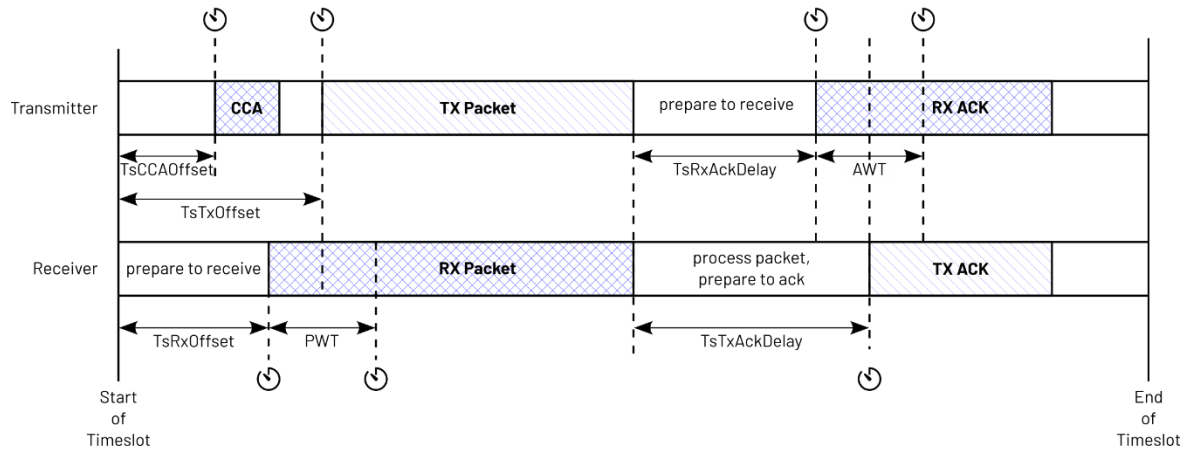

Figure 1 - TSCH timeslot template

Time is divided into timeslots. During a timeslot, a node knows exactly when to activate or deactivate its radio. Figure 1 presents an IEEE802.15.4 TSCH timeslot timing template for both transmitting and receiving nodes.

The packet transmission 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.

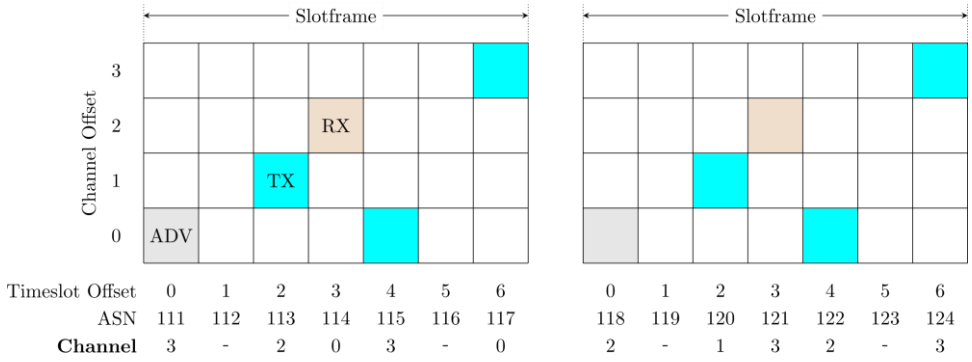

Figure 2 - IEEE802.15.4 TSCH Slotframes

An Absolute Slot Number (ASN) indicates the number of timeslots since the network was initiated. Figure 2 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 the Equation below, where *macHopSeqLen* is the number of elements in the *macHopSeq* list and % is the modulo operator.

$$\text{COUNTER} = \text{ASN} + \text{Channel Offset}$$

$$\text{Channel} = macHopSeq[\text{COUNTER} \,\%\, macHopSeqLen]$$

Using the Equation above ensures that, at each slotframe, a timeslot in each 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 to bring IPv6 to industrial low-power wireless networks (T. a. Watteyne 2015). 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 (LAKE) working group has been working on the standardization of a key exchange protocol for constrained network devices (Selander, Mattsson and Palombini 2024).

Hardware Assessment

This section presents the comparison of different microcontrollers to support the next generation of SmartMesh in the context of OpenSwarm. We start by comparing the evaluated microcontrollers and justifying the choice for the MAX32655. We then describe an in-depth assessment done to confirm the MAX32655 capabilities to run SmartMesh, experimenting with clock and timers' capabilities, wake up times and fine control of the BLE core. Finally, we present a circuit board containing not only the MAX32655, but the necessary components to guarantee that SmartMesh works correctly, as well as components required by the consortium members.

Microcontrollers Comparison

The first step was understanding the capabilities of the MAX326xx family of microcontrollers and comparing them with the LTC5800. Table 1 presents a simplified comparison of the BLE-enabled microcontrollers. Comparing all the microcontrollers:

Processor: all MAX326xx processors operate at faster clock than the LTC5800. Specifically, the MAX32655 RISC-V processor runs at a clock almost ten times faster than the LTC5800 and should be able to run the software stack.

Memory: all listed microcontrollers have larger Flash and RAM than the LTC5800 and should be able to accommodate the memory footprint required by SmartMesh.

Power Consumption: as general-purpose microcontrollers, we expect the power consumption to be higher for the MAX326xx family when compared to the LTC5800. Nevertheless, among them, the MAX32655 has the lowest power consumption.

Peripherals: all microcontrollers feature the basic peripherals for WSN applications, such as I2C, SPI and 1-wire. They also have AES, enabling hardware-accelerated encryption for the stack.

	LTC5800	MAX32655	MAX32666	MAX32690
Processor 1	Cortex-M3 @ 7.4 MHz	Cortex-M4 @ 100 MHz	Cortex-M4 @ 96 MHz	Cortex-M4 @ 120 MHz
Processor 2	-	RISC-V @ 60 MHz	Cortex-M4 @ 96 MHz	RISC-V @ 120 MHz
Flash	512 kB	512 kB	1 MB	3.25 MB
RAM	72 kB	128 kB	560 kB	1 MB
Radio	IEEE 802.15.4	BLE 5	BLE 5	BLE 5
Power Consumption	Active: 1.3 mA Radio TX: 5.4 mA Radio RX: 4.5 mA Sleep: 1.2 µA	Active: 2.4 mA Radio TX: 4.2 mA Radio RX: 4.1 mA Sleep: 7.1 µA	Active: 3.3 mA Radio TX: 4.3 mA Radio RX: 3.3 mA Sleep: 8.6 µA	Active: 9.2 mA Radio TX: 7.0 mA Radio RX: 4.8 mA Sleep: 23.3 µA

Table 1 - Microcontrollers comparison

Additionally, MAX32655's timer peripherals can be driven by the 32 kHz crystal oscillator, as it can be seen in Figure 3. This enables precise timing for TSCH while keeping low power operation.

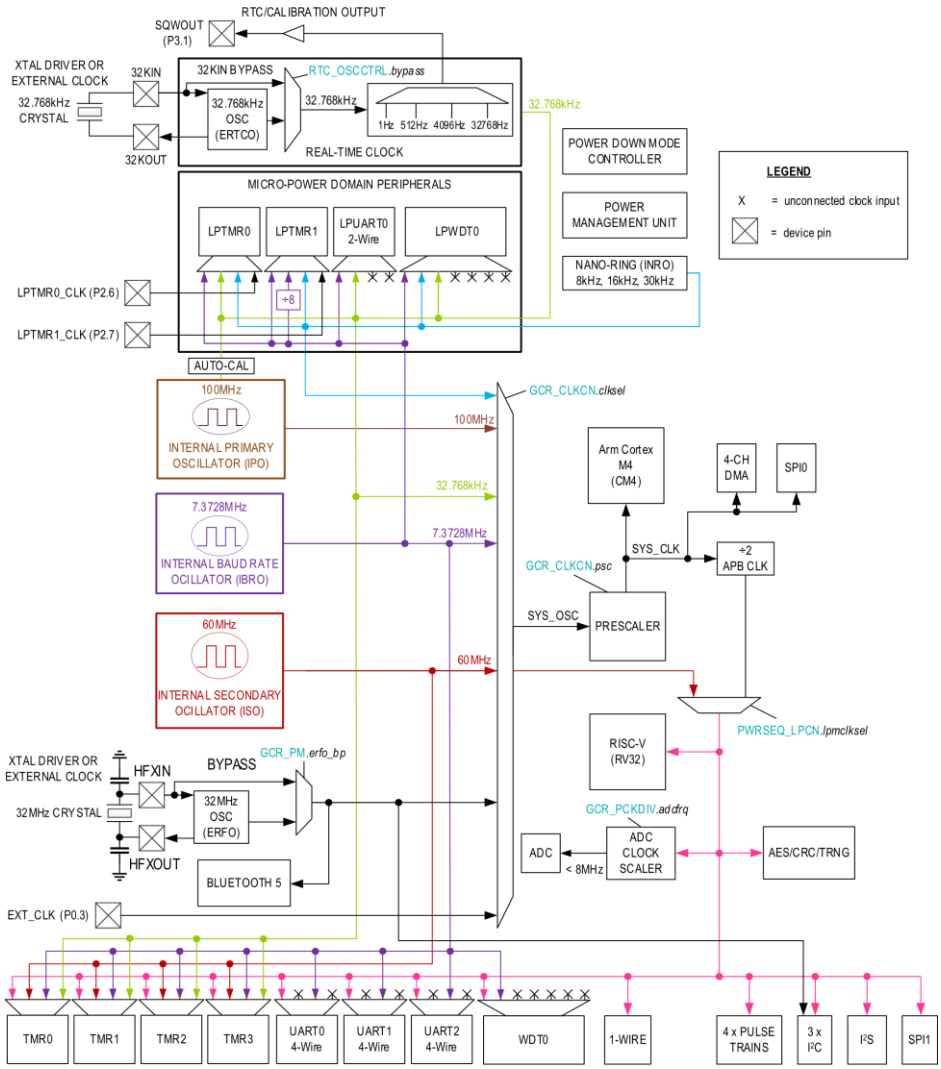

Figure 3 - MAX32655 clock scheme

MAX32655 Assessment

Among the proposed options in the MAX326xx family, the MAX32655 seems to be the most suitable, as it is comparable to the LTC5800 in terms of Flash and power consumption, but with an additional core and RAM. However, we still needed to evaluate it via experiments to confirm its feasibility to run SmartMesh. For that, we needed to answer some questions:

Can we schedule radio operations at specific times and wake up quickly enough?

SmartMesh is built on top of the IEEE802.15.4e MAC layer and uses Time-Slotted Channel Hopping (TSCH). Because of that, a timer is required to notify the processor at each event in the timeslot.

The initial idea was to use a timer (for example TMR0, as shown in Figure 3) to schedule the events inside a timeslot. However, to ensure low power operation, the microcontroller will be in sleep mode if no operation is scheduled. Because of that, the microcontroller must wake up in less than 10 µs after receiving an interrupt.

The MAX32655 has four different sleep modes, called SLEEP, LOW-POWER, MICRO-POWER and STANDBY, each one with different levels of power consumption and features. Preferably, we want to use the STANDBY mode, as it has the smallest power consumption.

To validate the expected wake-up times specified by the MAX32655 datasheet, we measured the wake-up times for the different sleep modes by putting the microcontroller to sleep and configuring a button to wake it up. After waking up, it turns an LED on. Using an oscilloscope, we can measure the time difference between the button press and LED becoming on, as can be seen in Figure 4.

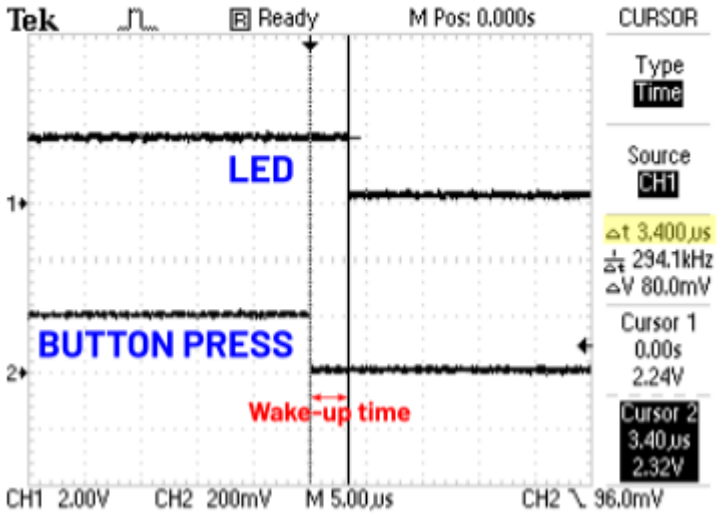

Figure 4 - Wake up time measurement

Table 2 compares the expected wake up times according to the MAX32655 datasheet. The times are below 10 µs, confirming that, in this criteria, the MAX32655 is suitable.

Mode	Datasheet	Measured (Cortex-M4, 100 MHz)	Measured (RISC-V, 50 MHz)
SLEEP	0.80 µs	0.50 µs ± 0.00 µs	5.82 µs ± 0.00 µs
LOW-POWER	6.08 µs	3.96 µs ± 1.72 µs	4.25 µs ± 0.01 µs
MICRO-POWER	12.40 µs	4.46 µs ± 1.92 µs	10.38 µs ± 0.01 µs
STANDBY	14.70 µs	5.00 µs ± 2.79 µs	9.851 µs ± 0.01 µs

Table 2 - Comparing wake up times for different sleep modes

Can we access the BLE radio directly?

The MAX32655 BLE examples show how to interact with the radio via a BLE stack. However, in this case, we wanted to implement the SmartMesh stack on top of the BLE physical layer, which means accessing the radio directly without using the default BLE stack.

Direct access to the BLE radio means that we must be able to:

- Send bytes to the peripheral to be transmitted.
- Get received bytes from the peripheral.
- Schedule the exact time for the radio to begin transmission.
- Schedule the exact time for the radio to begin reception and configure a timeout.
- Choose in which channel the operation will take place.

Additionally, as we are using a BLE 5 radio, we must be able to configure the mode of operation: 1 Mbps, 2 Mbps, 500 kbps or 125 kbps.

After thorough study of the BLE stack API, we confirmed that all the above can be achieved. To validate it, we used a spectrum analyser to confirm that the mode had been correctly configured and to verify that the frame duration was correct according to the BLE standard. Figure 5 shows the spectrum analyser readings for a transmission

of 127 bytes using the 1 Mbps mode. We verified that the frame duration was correct, equal to 1080 µs, corresponding to 127 bytes of data, 1 byte for preamble, 4 bytes for the access address and 3 bytes for the CRC at the end.

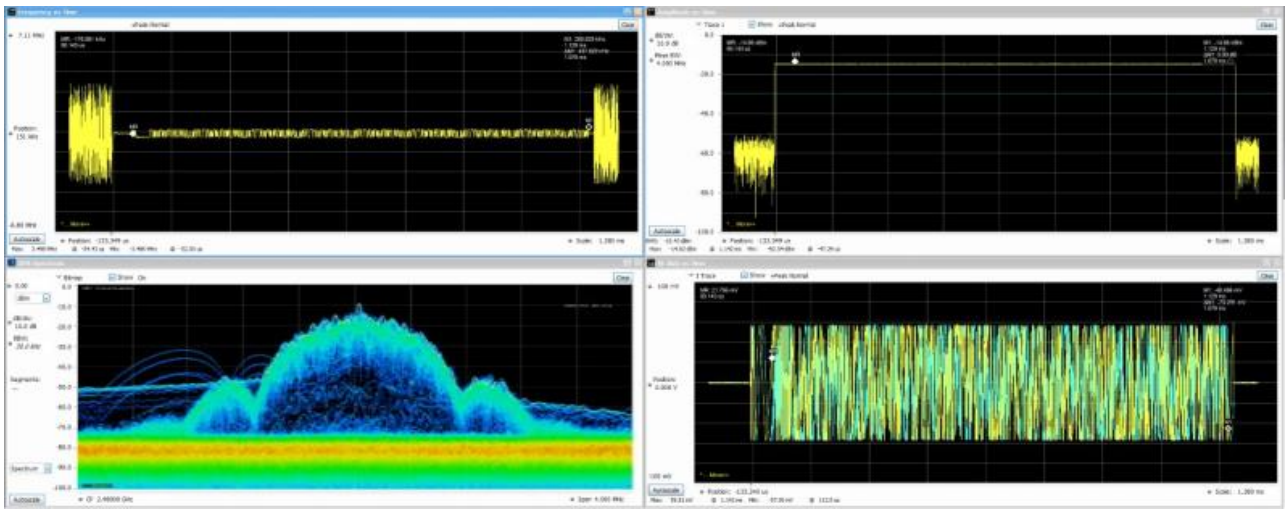

Figure 5 - BLE transmission frequency, amplitude, spectrum, and IQ values

To verify that the frame schedule time was correct, we connected two MAX32655 via a GPIO for synchronization and schedule operations at 2120 us and 7380 us after the beginning of a timeslot. Figure 6 shows the serial port readings for the experiment, confirming that the radio transmitted at the correct schedule time.

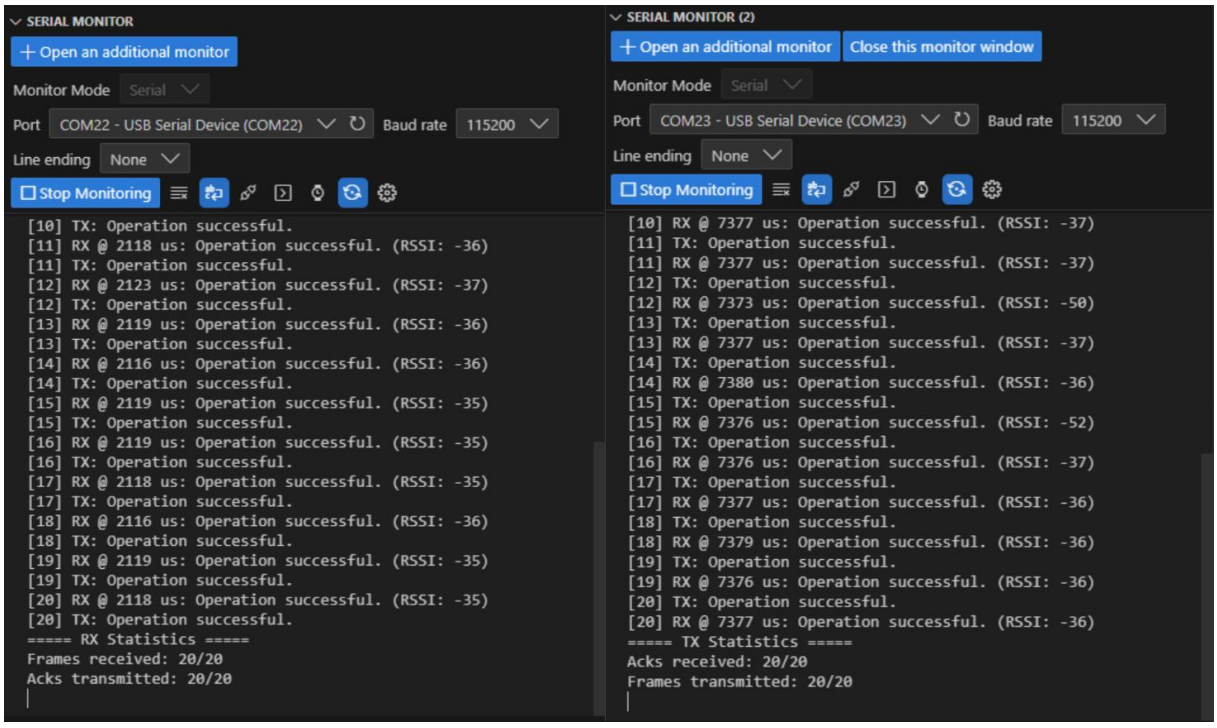

Figure 6 - BLE radio scheduling

Can we achieve low drifts to maintain the network synchronization?

To be power efficient, SmartMesh nodes are synchronized to within tens of microseconds. This way, nodes know exactly when to turn their radios on to transmit or listen to a packet. Because of that, there is a heavy requirement on the maximum drift allowed on the crystal oscillators.

To achieve low drifts and maintain network synchronization, we evaluated different MEMS resonators and crystals to drive the system. In this situation, temperature also plays an important factor, as changes in temperature modify the drift profile of crystals. Because of that, the MAX32655 must be used in conjunction with a temperature compensated crystal oscillator (TCXO).

The proposed TCXO is the MAX31328, a low drift (±3.5 ppm), temperature compensated RTC with integrated crystal. This will allow low power operation with a 32 kHz crystal while still achieving low drifts and keeping the network synchronization. Additionally, a simple 32 MHz crystal will be used to drive the BLE radio internally.

Final PCB

Figure 7 illustrates a high-level block diagram of the proposed PCB, which accommodates all the required components to make SmartMesh run on the MAX32655. To facilitate adoption by the consortium members and OpenSwarm users, the PCB will follow the Feather form-factor standard, as already used by several boards in the market. One example is the MAX78000FTHR board, shown in Figure 8. This allows quick prototyping and integration in existing projects, as the board dimensions and pinout are standardized.

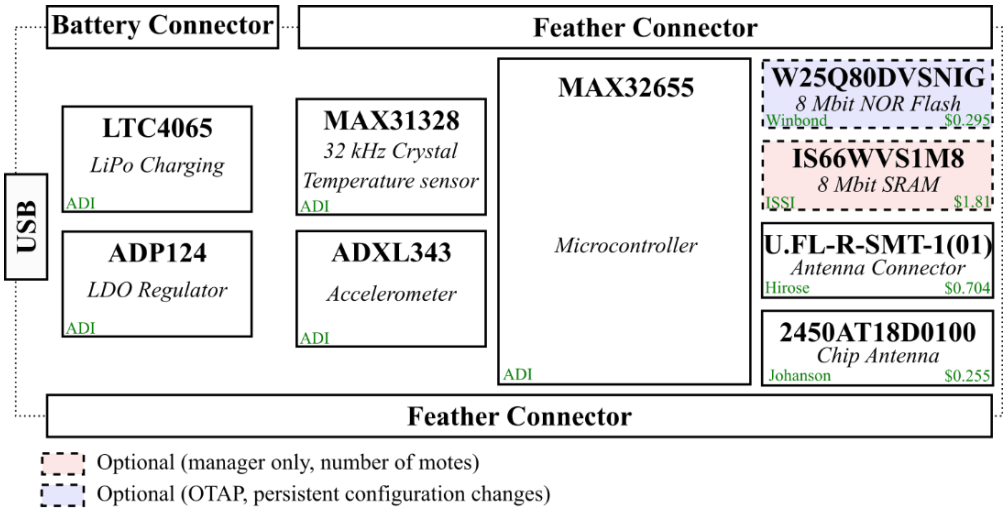

Figure 7 - Proposed PCB block diagram

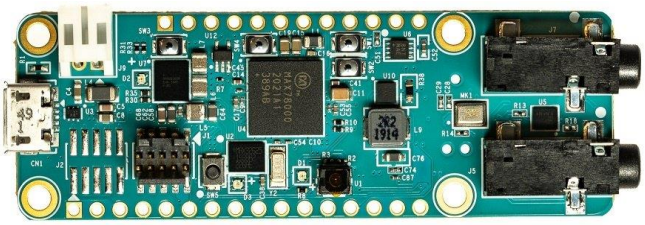

Figure 8 - Feather board example

The ADP124 regulator and LTC4065 LiPo charging IC allows the user power the board either from a USB connection or a battery. This facilitates the development phase as the user can operate the board via the USB connection and, later in production, switch to a battery seamlessly.

The U.FL antenna connector, together with a chip antenna, gives flexibility to the user to decide which antenna to use for every specific application, while the 10-pin Cortex Debug Connector allows connection to a debugger during the development phase.

Optionally, the user can add additional RAM memory to the board on the SmartMesh manager to allow for larger networks, as already done by the LTC5800. Furthermore, a Flash memory can also be added to allow specific configuration parameters to be saved in non-volatile memory.

Additionally, SmartMesh exposes a serial API via UART pins. We developed an adapter board with a simple FTDI USB-Serial converter. This way, the user can connect the

Feather board to the adapter board and connect to a PC and use the SmartMesh SDK[4], a Python library running on a PC, to interact with the network.

Figure 9 and Figure 10 show the finalized board layouts. Currently, the boards are being manufactured to be tested and then, made available to the consortium members.

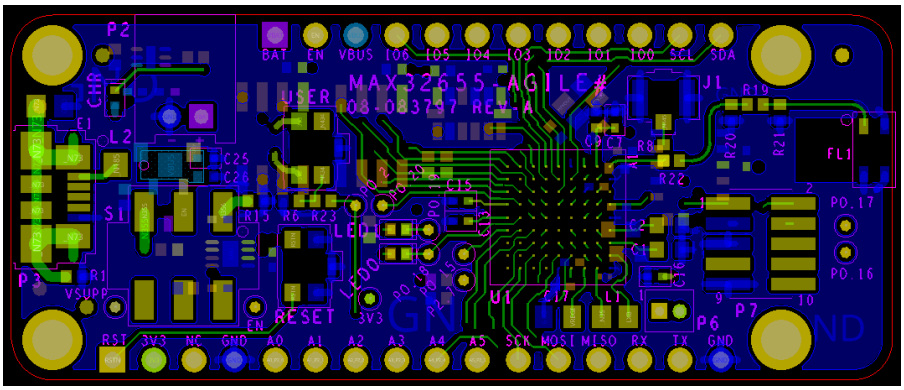

Figure 9 - SmartMesh Feather board layout

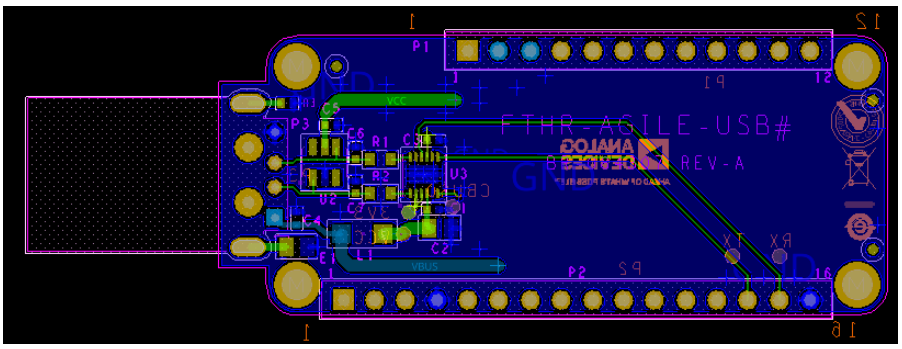

Figure 10 - SmartMesh adapter board layout

[4] https://dustcloud.atlassian.net/wiki/spaces/SMSDK/overview

Software

With the new microcontroller chosen, we can start porting the platform-dependent software part, which includes the internal SmartMesh drivers and lower MAC layer. We first present the architecture of SmartMesh, showing the components that exist around the stack. We then talk about µC/OS-II, the RTOS used in SmartMesh, and the effort to port it to Linux to expedite development of new features. Afterwards, we discuss the implementation of TSCH on the MAX32655, which we call Lower MAC Layer. Finally, we present the implemented IPC library to allow other OpenSwarm components to interact with the stack. This marks an important milestone, as for the first time, SmartMesh was ported to a different microcontroller.

Architecture

Figure 11 is a high-level software block diagram of SmartMesh. Users 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 SmartMesh local API, facilitated by the On-Chip Software Development Kit (OCSDK[5]) 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.

[5] https://dustcloud.atlassian.net/wiki/spaces/OCSDK/overview

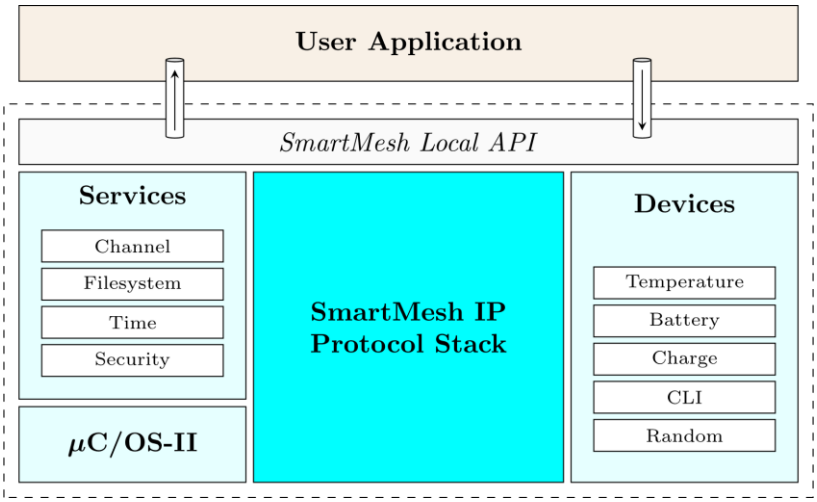

Figure 11 - SmartMesh software block diagram

Inside the protocol stack, the MAC layer can be subdivided into upper and lower MAC layer. In the context of porting SmartMesh, we are interested in changing the lower MAC layer, that is, code which interacts with the radio.

RTOS

A Real-Time Operating System (RTOS) is a type of Operating System (OS) intended to be used in environments where events have critically defined time constraints. This means that critical tasks must be completed within a time limit, as missing a deadline for hard real-time systems might lead to system failure. RTOSes are used for countless applications, including industrial, flight control, airline traffic systems and robots, as these applications usually need a guarantee of a soft or hard timing performance.

SmartMesh uses µC/OS-II, a portable, scalable, preemptive, real-time, multitasking kernel written in ANSI C and a small portion of assembly language to adapt to different processor architectures (Labrosse 1998). It was certified by the FAA, meeting the rigorous requirements of RTCA DO-178B. It was demonstrated through documentation and testing that the software is both robust and safe beyond the aerospace domain, including medical, industrial and transportation domains.

µC/OS-II supports preemptive multitasking, up to 255~tasks with one task at each priority level. Imperative structures like semaphores, mutexes, event flags, mailboxes and queues are supported, as well as software timers and deadlock prevention. Figure

12 details the file structure of the source code, showing all the implemented functionalities and clear separation between processor-independent and specific code.

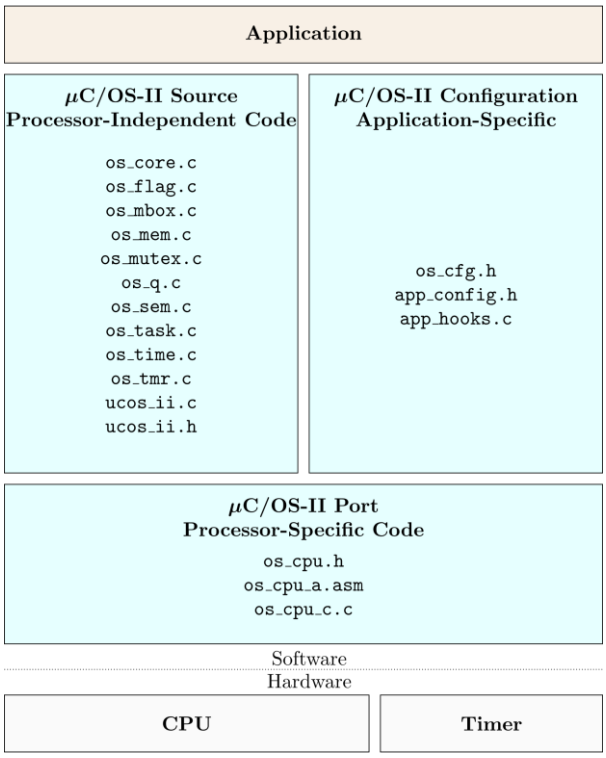

Figure 12 - µC/OS-II internal file structure

Besides performance, usage in safety critical system and memory footprint, license cost is also a determinant factor when choosing the RTOS for a given application. Originally, µC/OS-II required a commercial license, which imposed a cost for users wishing to develop products on top of it. However, the company behind µC/OS-II made a substantial change after considering the current software development environment, and the µC/OS-II components are currently offered under a permissive, open-source Apache 2.0 license model, allowing developers to use the components at no cost[6].

For the MAX32655, we upgraded µC/OS-II from version 292 to the Apache-Licensed version 293, facilitating integration within the OpenSwarm project.

6 https://weston-embedded.com/micrium-licensing

Simulator

When developing embedded applications, debugging, and benchmarking becomes a laborious task. The use of simulators can increase productivity when developing an application on top of an RTOS, as the developer can save time not having to flash the microcontrollers every time a small change is made to the code base.

To expedite SmartMesh development in a new platform, we first ported µC/OS-II to Linux, where it can run natively as a process. This enables users with the possibility to debug µC/OS-II applications on a Linux PC without the need to continuously reflash the board, accelerating development time, and reducing costs.

With the port, we can run SmartMesh on a Linux PC and, before downloading it to the MAX32655, create a virtual network and validate the stack functionalities. Figure 13 shows an example of a virtual network running on Linux, where we can simulate the RF channel between pairs of nodes and see the network forming in the specific scenario.

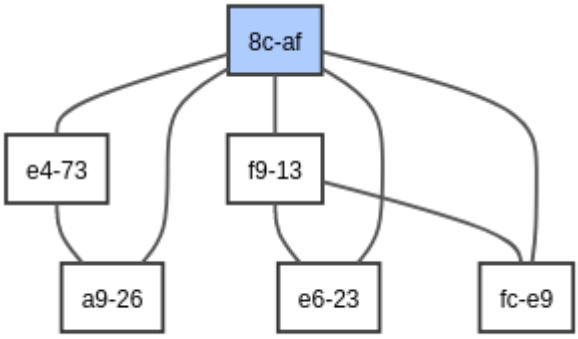

Figure 13 - Topology of a simulated network running on a Linux PC

Lower MAC Layer Implementation

The Lower MAC Layer implements the TSCH protocol and is responsible for executing the timeslot shown in Figure 1. As a starting point, we implemented it based on OpenWSN, a repository for open-source implementations of protocol stacks based on IoT standards (Watteyne, et al. 2016).

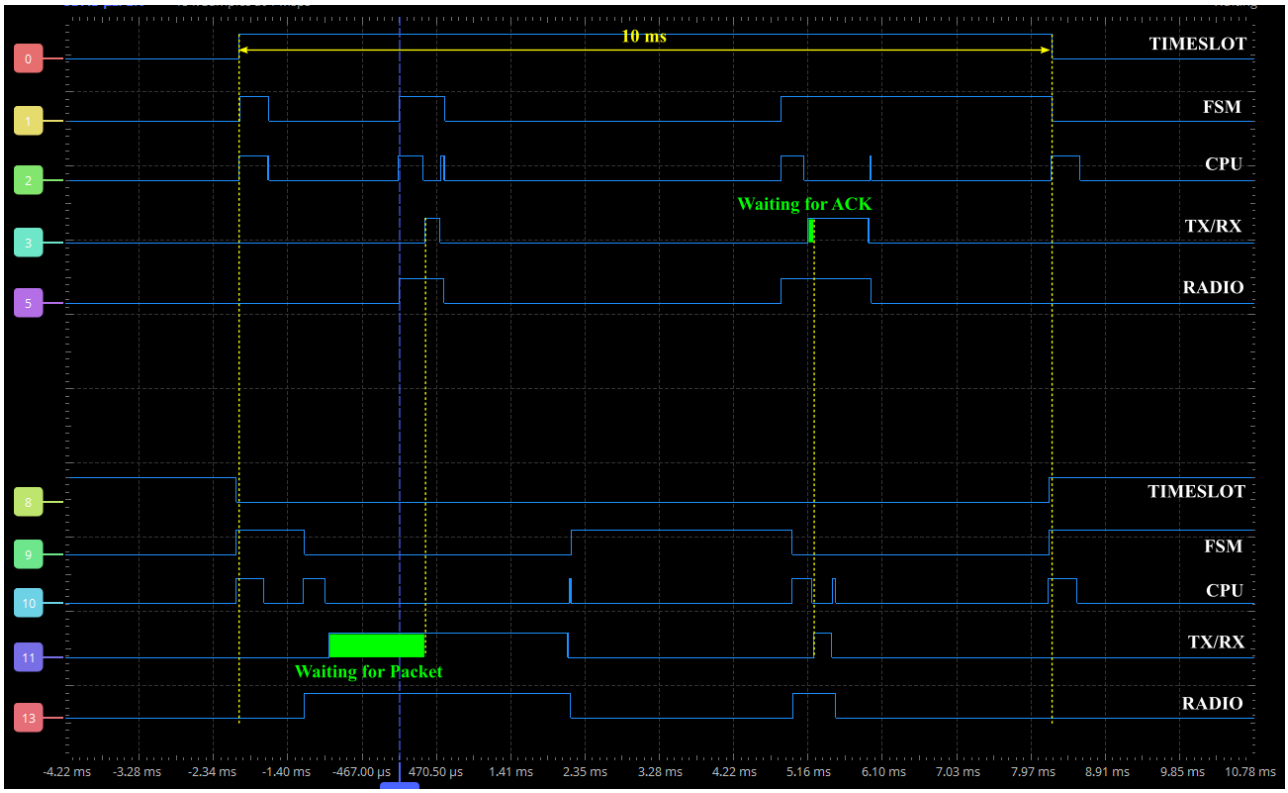

Figure 14 - Final TSCH state machine for both transmitter and receiver

Figure 14 shows the different events that occur throughout a single TSCH timeslot in a logic analyser. To validate the implementation, we configured different GPIOs to toggle at specific events in the timeslot. In the figure, we can see a standard 10 ms timeslot for two motes, one transmitting (top) and the other receiving (bottom). As per TSCH, the top node waits for an ACK after transmitting a packet.

The TX/RX lines show when the packets are being transmitted, and we can see that, for both the data packet and the ACK, the radios are configured correctly to transmit and receive at the configure moments according to the TSCH template.

IPC API

As we run SmartMesh on the RISC-V, we need to create an IPC API to enable the user to interact with the stack. Figure 15 shows how an application running alongside SmartMesh on the MAX32655 is distributed between the cores. The user application runs on the Cortex-M4 and can directly access the peripherals. The protocol stack runs

on the RISC-V and has exclusive control of the BLE radio. There is also a channel between the cores to exchange data.

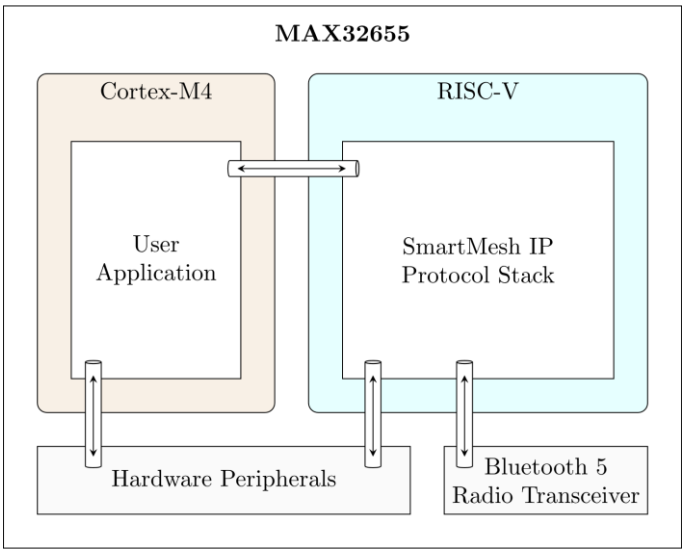

Figure 15 - Software diagram of SmartMesh running alongside the user application

Originally, to interact with the network using the LTC5800, the user would need to use the OCSDK, which uses channels. We adapted the channel implementation to support the exchange of messages between the cores by leveraging MAX32655's mailboxes and semaphores capabilities. A mailbox is simply a shared memory area, from which both cores can read and to which both cores can write. A semaphore is a hardware peripheral which allows one core to notify the other core via interrupts. Therefore, mailboxes and semaphores can be used in conjunction to send a message to the RISC-V by the following steps:

- The Cortex-M4 writes some data to the mailbox
- The Cortex-M4 uses a semaphore to notify the RISC-V about the data
- The RISC-V, after being notified, reads the data from the semaphore
- Then, the RISC-V notifies the Cortex-M4, so it knows the message has been read
- Finally, the Cortex-M4 is allowed to send more messages

The sequence diagram in Figure 16 shows the implementation of the IPC library, exemplifying the message exchange above.

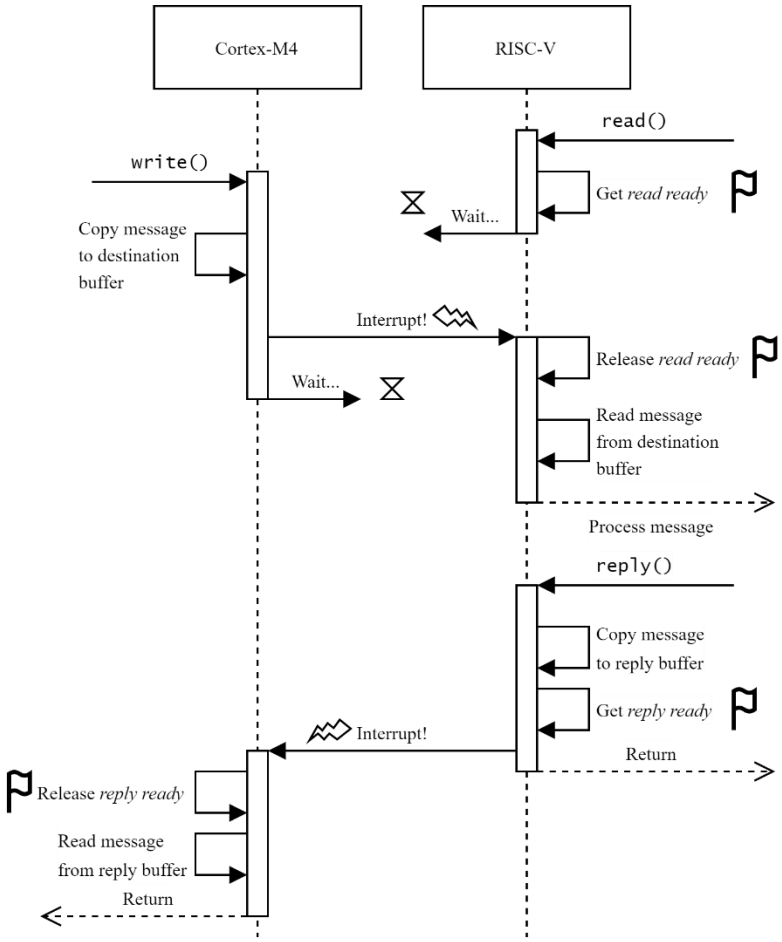

Figure 16 - Example of IPC message exchange

To improve the programming experience and expedite user development, we abstracted the channel details away from the user and wrapped the Local Interface, the Channels and the IPC Library in a new API, which contains basic functions to interact with the network (for example, join a network, send and receive packets, configure MAC address, etc). The new API is fully documented both as a set of web pages and as a PDF document, as in Figure 17.

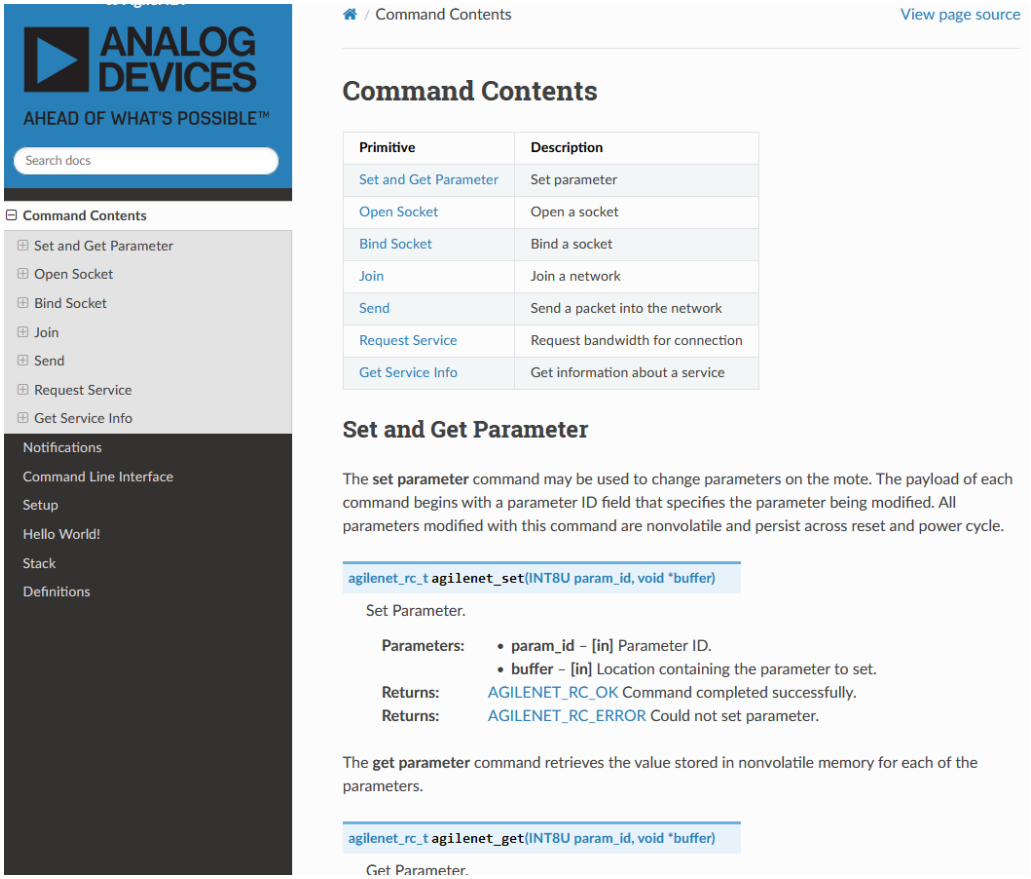

Figure 17 - New API to interact with the network

The First Prototype

To validate the port, we ported SmartMesh targeting the MAX32655FTHR board, an affordable and readily available evaluation board for the MAX32655. Thanks to the clear separation between the application, network, MAC, and driver layers, the network and application layers were reused to maintain existing functionalities. Meanwhile, the lower MAC and driver layers have been implemented from scratch with more than 15,000 new lines of code to date.

Figure 18 shows the setup used during development, where two MAX32655FTHR boards (one for the SmartMesh manager and the other for a SmartMesh mote) were used in conjunction with an ADALM2000 logic analyzer for debugging and benchmarking.

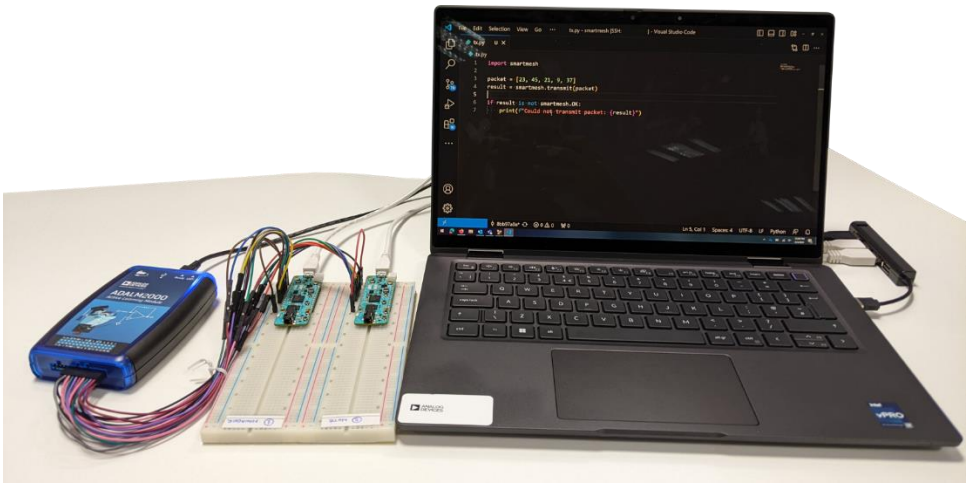

Figure 18 - Two MAX32655FTHR boards running SmartMesh connected to a PC

Summary

This successful port to the MAX32655 microcontroller paves the way for a new generation of SmartMesh devices. The affordability and availability of the MAX32655 make it an ideal choice for widespread adoption. The clear separation between application, network, MAC, and driver layers allowed for the reuse of existing functionalities while implementing new lower MAC and driver layers from scratch. This enables seamless interaction between the application and the network through the new SmartMesh local API on top of the IPC library.

Mobility and Latency Predictability

This section presents some techniques used to enable mobile nodes and to support latency predictability within a SmartMesh network. In a mobile network, nodes may temporarily leave the coverage area, resulting in their disconnection from the network. Upon re-entering the coverage area, these nodes must rejoin the network, a process that can be time-consuming depending on the network parameters. We explain the SmartMesh join procedure, highlighting the configurable parameters and their impact on join duration, and explore the methods employed to enhance join times and their implementation within SmartMesh. Additionally, we introduce the concept of SmartMesh Backbone mode, which is used to reduce latency, and provide an initial survey of different scheduling algorithms used in TSCH networks. This literature review is a crucial step before designing and implementing new scheduling strategies to support latency predictability. As Tasks 5.4 and 5.5 have already kicked off, the actual implementation work has been deferred to prioritize the design and manufacturing of the new SmartMesh boards in Figure 9 and Figure 10.

SmartMesh Join Procedure

When a SmartMesh mote wants to join a network, it needs to hear advertisement messages sent by the SmartMesh manager or by another SmartMesh mote already in the network. Once it hears an advertisement, it synchronizes to the network and starts the joining process with the manager. This can be done even when the manager is multiple hops away from the joining mote. Finally, when the joining process is finished, the mote is said to be in the **operational** state. It has been assigned some base bandwidth, allowing it to send infrequent traffic to the manager,

The speed at which a mote joins is determined by some components:

- Advertising rate: the rate at which motes in the network advertise. The user has little control of this other than mote density, or turning advertisement off explicitly at the manager
- Join duty cycle: how much time a searching mote spends listening for a network versus sleeping. This is fully configurable by the user.
- Mote join state machine timeouts: The user has no control over these values.
- Downstream bandwidth: affects how quickly motes can move to the operational state.
- Number of motes: contention mong many motes simultaneously trying to join for limited resources slows down the join process because of collisions.
- Path stability: the physical stability of the RF medium; the user has little or no control over it.

Currently, the only user configurable parameter to increase join speed is the duty cycle, which incurs a trade-off between join speed and energy consumption. For example, if the duty cycle is 100%, the mote will spend a lot of power, as it will have its radio on for 100% of the time to listen for advertisements. Additionally, the advertisements are sent over 16 different physical channels, and the mote cycles through these channels to listen to advertisements.

BLE Advertisement

Wireless technologies targeting multiple-node networks, in which a node can join and leave the network, specify some type of join procedure. For example, Bluetooth specifies the connection process, where advertisements allow devices to broadcast their presence and capabilities to other devices. With this in mind, BLE 5 introduced several enhancements in the way it handles advertisements; it optimizes the connection process by using three primary advertisement channels, shown in Figure 19.

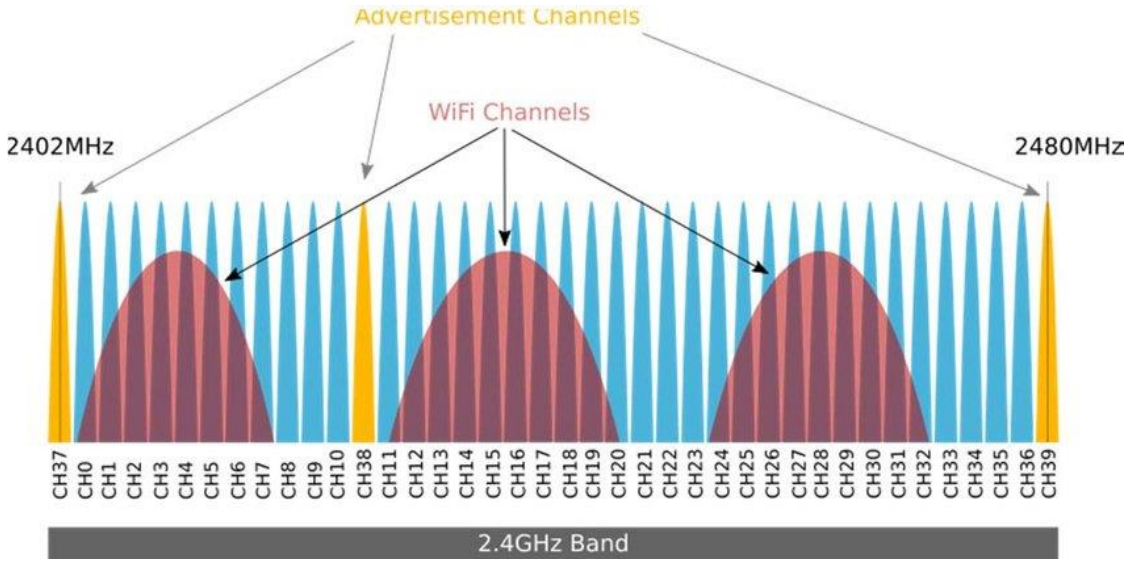

Figure 19 - BLE vs WiFi channels (Andrei Gheorghiu 2017)

BLE radios use 40 channels, each 2 MHz wide. Out of these, three channels (37, 38, 39) are designated as primary advertisement channels. These channels are strategically chosen to minimize interference from other wireless technologies co-existing in the 2.4 GHz ISM band, such as WiFi.

As only three channels are used to advertise, scanning devices can quickly cycle through them, leading to faster discovery and connection times and reduced energy consumption.

Proposed Join Procedure Improvements

SmartMesh IP Application Notes[7] details the trade-off between search time and average current consumption, stating that, if a mote is close to the network and can hear advertisements, a join duty cycle of 5% yields a join time of 3 minutes on average, while a join duty cycle of 25% yields a join time 30 seconds on average. On one hand, mobile nodes might be disconnected for considerable periods, and even 25% duty cycle can waste a lot of energy. On the other hand, waiting 3 minutes on average to join using 5% duty cycle might be unacceptable.

7 https://www.analog.com/media/en/technical-documentation/application-notes/smartmesh_ip_application_notes.pdf

Following the idea used in BLE, our first approach is to allow the configuration, by the user, of the number of channels to be used. For some applications, advertising on one channel might be sufficient, as in mobile scenarios the channel coherence time is short, and channel interference and multipath effects change quickly, allowing the mote to join. For other applications, the user might want to configure more advertisement channels to ensure enough frequency diversity.

The second approach impacts both the advertising rate and the downstream bandwidth by reducing the timeslot length. Reducing the timeslot length reduces the slotframe size, increasing the bandwidth and allowing advertisements to be sent more frequently.

The timeslot length in SmartMesh is limited by:

- The maximum packet length in 802.15.4, which is 127 bytes
- The maximum acknowledgement length in 802.15.4, which is 15 bytes

As the 802.15.4 raw data rate is 250 kbps, the time required to send a maximum length packet and acknowledgement is 4.064 ms and 480 µs, respectively. Assuming a standard TSCH timeslot, that is, using the values shown in Figure 1 according to the standard values presented in (Vilajosana 2020). Using the 1 Mbps BLE PHY present in the MAX32655, these values can be reduced to 1.016 ms and 120 µs, respectively, and a standard timeslot TSCH could be reduced from 10 ms to 6.592 ms.

Without taking the internal mote state machine timeouts into consideration, advertising on only 3 channels instead of 16 can yield a speedup of 5.33, while advertising on only 1 channel can yield a speedup of 16 (assuming perfect RF conditions). Additionally, reducing the timeslot length from 10 ms to 6.592 ms yields a speedup of 1.51. Even though the strategies were implemented during the project, these values represent theoretical estimations; in the future, experiments will be carried on confirming the actual speedups when compared to original SmartMesh.

SmartMesh Backbone Mode

The concept of the powered backbone in SmartMesh IP networks is designed to improve network performance, particularly in terms of reducing latency. It is a special slotframe configuration that enables low-latency communication within the network.

The user can enable two different modes:

- Upstream Backbone: When the upstream backbone is enabled, powered motes (those with a steady power source) listen for packets from their children in every slot. This means that any mote can transmit data to its parent at any time, reducing the average latency per hop to about 15 milliseconds. This is particularly useful for applications requiring low-latency alarms or real-time data collection.
- Bi-Directional Backbone: In this mode, the backbone supports both upstream and downstream communication. This is achieved by alternating slots for upstream and downstream traffic, ensuring that there is no collision between the two directions. The bi-directional backbone is ideal for applications that require fast call-and-response communication, such as remote control systems. It typically achieves round-trip latencies of about 60 milliseconds per hop.

This scheduling can benefit many applications, such as:

- Low-Latency Alarms: For networks where quick response times are critical, such as in industrial monitoring or safety systems, the upstream backbone ensures that alarms can be transmitted with minimal delay. This can achieve about 15 ms latency per hop for an additional 925 µA at routing devices.
- Call and Response: In scenarios where devices need to quickly acknowledge commands or data requests, the bi-directional backbone mode provides a reliable and fast communication path. This can achieve about 60 ms latency per hop for an additional 462 µA at leaf nodes and 925 µA at routing devices.
- General Low-Traffic Networks: Even in networks with low overall traffic, enabling the upstream backbone can reduce latency for all motes, especially those close

to the manager. This can achieve about 15 ms latency from one-hop motes without additional power at motes.

While the powered backbone significantly reduces latency, it does come with increased power consumption for the motes involved, particularly those that are powered and thus responsible for listening in every slot. Therefore, it is best suited for networks where some motes have a reliable power source.

Literature Survey

When designing a scheduling algorithm, we need to choose an objective function to optimize. This could be minimizing latency, overhead or energy consumption, as well as maximizing reliability or throughput. In our case, we need to understand the algorithms that minimize latency and energy consumption, and then understand how we can ally both to create a configurable trade off. As discussed in the last section, if all the timeslots are scheduled, each packet will be transmitted in less than a timeslot length, but this can incur in additional energy consumption. On the other hand, scheduling the minimum number of cells will save energy, but will maximize latency. We need to enable the users with a way to tailor these values to their applications, i.e., for a given power consumption requirement, we can predict the latency that can be achieved, and vice-versa.

Several strategies have been proposed to minimize latency:

- Low Latency Scheduling Function (LLSF): This strategy adopts a daisy-chaining approach for timeslots rather than selecting them randomly. When nodes schedule a timeslot from one node to another, LLSF daisy-chains the timeslots used in a multi-hop path. This method significantly reduces end-to-end latency by ensuring that data packets are forwarded promptly from one node to the next without unnecessary delays.

- Low-Latency Distributed Scheduling Function (LDSF): LDSF organizes the slotframe into smaller parts known as blocks. Each transmitter, based on its hop distance from the border router, selects the appropriate set of blocks, enabling

the automatic scheduling of retransmission opportunities. This approach minimizes end-to-end delay by effectively chaining blocks, ensuring that packets progress by one hop at the end of each block.

- Centralized Scheduling Algorithms: Algorithms like TASA (Traffic Aware Scheduling Algorithm) use centralized control to optimize the allocation of timeslots and channel offsets across the network. By leveraging global network information, these algorithms can minimize latency by ensuring that data packets follow the shortest and least congested paths.

Additionally, other strategies have been developed to minimize power consumption:

- Parameterized Adaptive and Autonomous Scheduling (PAAS): PAAS operates as an autonomous and distributed algorithm that dynamically adapts to traffic intensity, slotframe length, and reliability requirements. It minimizes energy consumption by adjusting the slot schedule based on the current traffic load, ensuring that nodes only wake up and communicate when necessary.

- Energy-Efficient Scheduling (EES): EES focuses on creating a schedule that maximizes energy efficiency by examining the energy consumption model for TSCH nodes. It employs a collective greedy strategy to assign each available frequency to the node that can achieve the maximum energy efficiency at that frequency, thereby reducing overall energy consumption.

- Swarm-based Energy Efficient Scheduling (TREE): TREE operates at the MAC layer independently of other layers, treating all incoming packets uniformly. It triggers transmission cell scheduling for a neighbor with a loaded queue and releases underused or interfering cells. This approach ensures that communication resources are scheduled based on the device's need for each neighbor, preventing excessive energy consumption.

To create a solution with configurable latency for a given power consumption target, we can integrate elements from both latency-minimizing and power-saving strategies:

- Adaptive Slot Scheduling: Implement an adaptive slot scheduling mechanism that dynamically adjusts the slotframe length and the number of active slots based on real-time traffic conditions and power consumption targets. This can be achieved by combining the daisy-chaining approach of LLSF with the adaptive nature of PAAS.
- Priority-based Scheduling: Introduce a priority-based scheduling algorithm that assigns higher priority to latency-sensitive traffic while ensuring that low-priority traffic is scheduled in a way that minimizes power consumption. This can be done by integrating the centralized control of TASA with the energy-efficient strategies of EES.
- Hybrid Scheduling Functions: Develop hybrid scheduling functions that incorporate both latency and energy efficiency metrics. These functions can use machine learning algorithms to predict traffic patterns and adjust the schedule dynamically, ensuring that the network meets both latency and power consumption targets.

Conclusion

This deliverable is part of WP2, which aims to develop the core OpenSwarm technologies related to orchestration of collaborative smart nodes, including networking, security, coordination and management. Specifically, this deliverable contributes to the dependable networking aspect.

We present how we achieved the port of SmartMesh to the state-of-the-art MAX32655, to be made widely available to the consortium and beyond. After a thorough assessment of possible microcontrollers to replace the original LTC5800, we developed new boards to meet the synchronization requirements of SmartMesh and implemented the TSCH hardware features of the LTC5800 in software. This process included porting the network stack to run on the RISC-V core, creating an inter-process communication (IPC) library, and developing a new user API, which allows easy integration by the consortium members. This transition not only reduced manufacturing costs but also enhanced the flexibility and scalability of the SmartMesh system.

This enables us with a solid base to work on the mobility and latency predictability aspects. For mobility, we propose the reduction of the number of advertisement channels and the reduction of the TSCH timeslot length, allowing faster join times for nodes leaving and re-entering the network. In terms of latency predictability, we perform a literature survey to understand the techniques that can be used.

Throughout the project, we prioritized the design and manufacturing of the new SmartMesh boards, which will enable the consortium members with the updated version of SmartMesh for both the Implementation and Verification of OpenSwarm as well as the Proof-of-Concept-based Validation phases.

Appendices

This section presents the papers developed throughout Task 2.1.

Running MicroPython on the MAX32655 with SmartMesh

Authors: Luiz Sampaio, Kate O'Riordan, Dara O'Sullivan, Brian Coffey, Lance Doherty, Thomas Watteyne

Status: Accepted – to be presented at the Conference of the IEEE Industrial Electronics Society (IECON) in Chicago, IL, USA, 3-6 November 2024

To demonstrate the separation between user application (Cortex-M4) and network stack (RISC-V) on the MAX32655, we show how the Cortex-M4 can run a MicroPython interpreter for faster prototyping.

MicroPython[8] 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.

8 https://micropython.org/

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 Figure 18. These off-the-shelf boards come equipped with a debugger for quick code downloading and testing and offer a serial connection for message printing. We connected an ADALM2000 to the boards and used it as a logic analyser.

Leveraging MicroPython's *embed* port, we create a periodic task to execute the simple MicroPython code below, both as a script and as bytecode. The code is used to transmit a packet through a SmartMesh network. Upon the task's awakening every second, a GPIO pin is brought high and recorded by the logic analyser. 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.

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

smartmesh.transmit(packet)
```

We measure the processing time to transmit packets of different lengths, as illustrated in Figure 20 - Processing time to send a packet. 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.

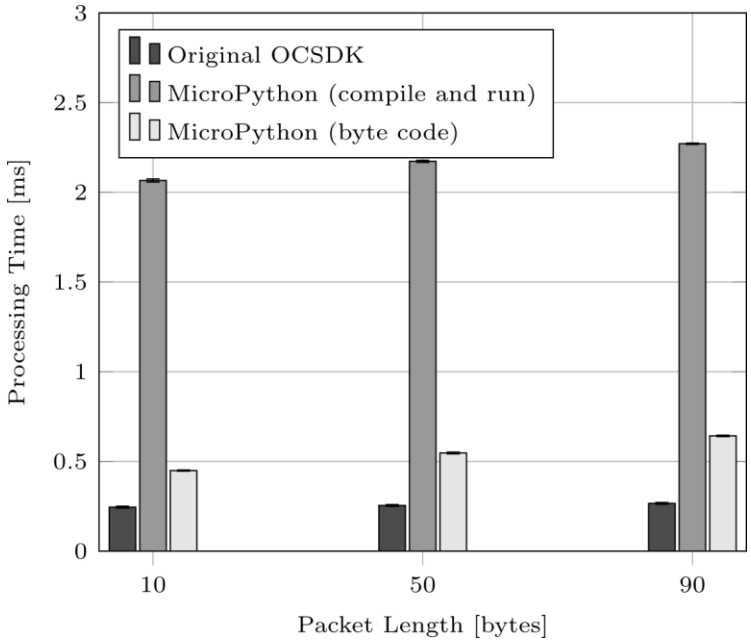

Figure 20 - Processing time to send a packet

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 3. In the context of the experiment, all memory is statically allocated, therefore the *size* command yields a reasonable estimate of RAM usage.

	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

Table 3 - Flash and RAM Memory Usage Comparison

As expected, the additional MicroPython code, compiled solely with the simple *smartmesh* Python 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.

Running µC/OS-II on Embedded Linux in Production

Authors: Luiz Sampaio, Kate O'Riordan, Dara O'Sullivan, Brian Coffey, Lance Doherty, Thomas Watteyne

Status: Submitted to the ACM International Conference on the Foundations of Software Engineering (FSE), in Trondheim, Norway, 23-27 June 2025

As part of the simulator created to expedite development, we conducted a measurement campaign to evaluate the performance of the port on Embedded Linux in terms of interrupt response, interrupt recovery and context switch latency.

For this experiment, we create a button IRQ and two tasks. The button IRQ interrupts the Idle Task when a button is pressed and posts a semaphore to make Task A ready to run, which posts another semaphore to make Tasks B and C ready to run, as show in Figure 21. A GPIO pin is configured and toggled at every event of interest to measure three performance metrics:

- **Interrupt Response**, defined as the time between the reception of the interrupt and the start of the user code that handles it
- **Interrupt Recovery**, defined as the time required for the processor to return to the interrupted code or to a higher priority task if the interrupt service routine made such a task ready
- **Context Switch Latency**, herein expressed by the time for a complete context switch triggered by a semaphore

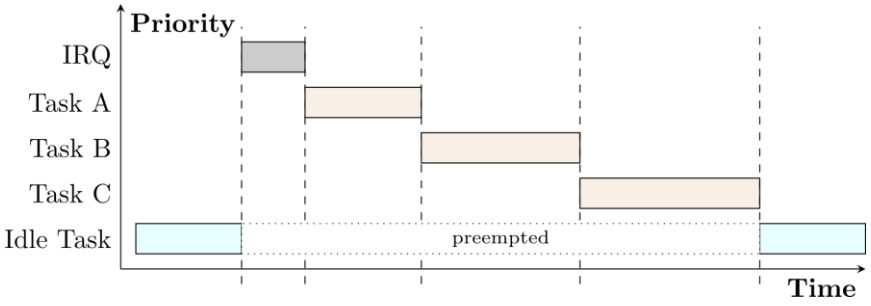

Figure 21 - Experiment timeline

The experiments are carried out on an STM32F769I Discovery board, which is based on an ARM Cortex-M7 microcontroller running at 200 MHz. The board features a button connected to a GPIO that can be configured as an interrupt, and an SD card connector to hold the Linux image.

In order to create a lightweight Linux distribution for the STM32F769, we used Buildroot[9], a tool to generate embedded Linux systems through cross-compilation. This allows us to compile a Linux kernel only with the necessary Linux subsystems for the experiments, minimizing the final flash image size.

The code snippet below shows the button interrupt service routine on Linux. In this case, the button is used as a key leveraging the Linux input event subsystem. The *fd_real* file descriptor is obtained by opening the */dev/input/event0* file, and therefore one should read the number of bytes in the *input_event* structure.

```
void* button_handler(void* unused)
{
#if OS_CRITICAL_METHOD == 3u
      OS_CPU_SR cpu_sr;
#endif

      while (1) {
            read(fd_read,
                  &ev, sizeof(struct input_event));
            OS_ENTER_CRITICAL();
            OSIntEnter();
            OS_EXIT_CRITICAL();

            lseek(fd_read, 0, SEEK_SET);

            OSSemPost(interruptSem);

            OSIntExit();
      }
      return NULL;
}
```

For each performance metric, the experiment is repeated 10~times and measured by the ADALM2000 logic analyser at 1~M samples per second.

The measured times are shown in Table 4 and Figure 22.

[9] https://buildroot.org/

		Mean	Maximum
Bare Metal	Interrupt Response	693.8 µs	696.1 µs
	Interrupt Recovery	13.5 µs	14.8 µs
	Context Switch	9.4 µs	10.0 µs
Linux	Interrupt Response	6,413 µs	6,752 µs
	Interrupt Recovery	914.8 µs	931.1 µs
	Context Switch	391.1 µs	396.5 µs

Table 4 - Measured latencies for µC/OS-II running on STM32F769

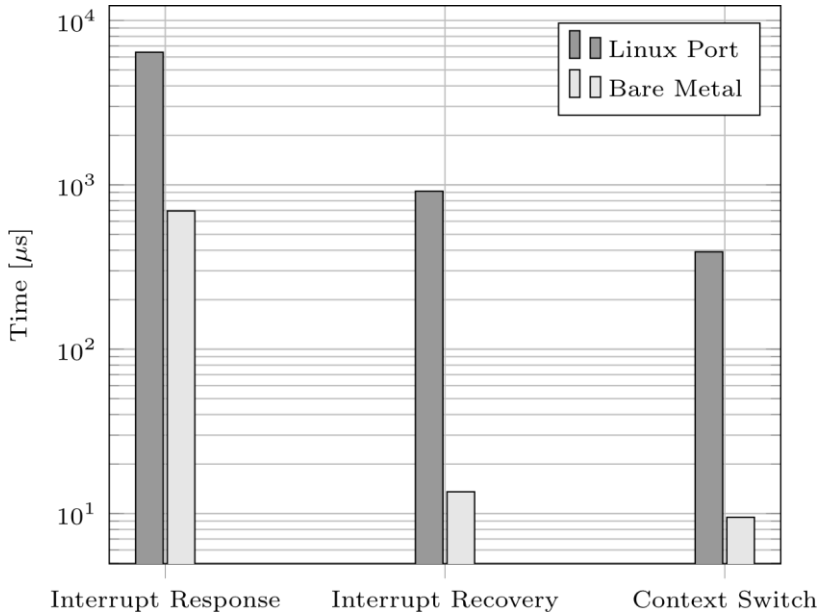

Figure 22 - Mean values for the measured latencies

We see that, as expected, the latencies for µC/OS-II running on Linux are between one and two orders of magnitude larger when compared to bare metal. The interrupt response overhead on Linux contains not only the interrupt latency of the CPU, but also the overhead added by the Linux GPIO and input event subsystems, which parse the GPIO line and converts it to the configured key before notifying the user application. For the interrupt recovery and context switch, while the bare metal version only triggers the

PendSV exception to perform context switch, the Linux version adds its own scheduler and pthread library overheads.

Nevertheless, the standard deviation and the maximum latencies are close to the mean, which indicates that µC/OS-II can be used over Linux in production, as the worst-case latency is predictable, when the average-case latencies are acceptable for a given application. The results show that it is feasible to run µC/OS-II over Linux, which accelerates the development and debugging phases. As an example, the STM32F769 Discovery board contains an Ethernet connector, and code can be recompiled and downloaded to the board running Linux on-the-fly via SCP or TFTP for tests.

Table 5 shows the flash and RAM usage for the bare metal and the Linux system.

		Bare Metal	**Linux**
Flash Usage	Application	15.1 kB	86.2 kB
	Kernel	0 kB	1.6 MB
	RootFS	0 kB	2.8 MB
	Bootloader	0 kB	378 kB
RAM Usage	Application	14.7 kB	56.7 kB

Table 5 - Flash and RAM memory usage comparison

The sample applications use static allocation only, therefore the toolchain *size* utility gives a realistic estimation of RAM usage. As expected, the default image generated by Buildroot uses two orders of magnitude more flash than the bare metal version, as the latter does not need a bootloader, a Linux kernel and a basic root file system. Nevertheless, the custom Linux image occupies less than 2 MB, which is much less than general purpose Linux distributions. Additionally, as the application size grows, the memory overhead for the libraries linked against it (such as pthread and Linux libc), the Linux kernel and the bootloader become smaller in relative terms. This confirms that it

is possible to create a lightweight Linux image targeted at specific applications for devices with only a few megabytes of flash.

The use of µC/OS-II on Linux can leverage its entire ecosystem of supported devices. This means that a user does not need to re-implement a device driver if it is already implemented by the Linux kernel. Moreover, Linux also implements a range of file systems and network protocols natively, and developers can tailor the Kernel to contain only the necessary subsystems for the specific application, saving development time. Finally, open-source libraries targeting Linux can be easily integrated, and Linux programs such as shells, simulators and performance tools can be used to support, test and profile the µC/OS-II application on-the-fly.

Bibliography

Andrei Gheorghiu, Valentin Iordache, Angel Ciprian Cormos. 2017. "Analysis of handshake time for Bluetooth communications to be implemented in vehicular environments." *40th International Conference on Telecommunication and Signal Processing.* Barcelona: IEEE.

Labrosse, Jean J. 1998. *MicroC/OS-II.* R&D Books.

Selander, Göran, John Preuss Mattsson, and Francesca Palombini. 2024. *Ephemeral Diffie-Hellman Over COSE (EDHOC).* Internet-Draft, Internet Engineering Task Force.

Vilajosana, Xavier and Watteyne, Thomas and Chang, Tengfei and Vučinić, Mališa and Duquennoy, Simon and Thubert, Pascal. 2020. "IETF 6TiSCH: A Tutorial." *IEEE Communications Surveys & Tutorials* 595--615.

Watteyne, Thomas and Palattella, Maria Rita and Grieco, Luigi Alfredo. 2015. *RFC7554 - Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement.* Internet-Draft, Internet Engineering Task Force.

Watteyne, Thomas, Vlado Handziski, Xavier Vilajosana, Simon Duquennoy, Oliver Hahm, Emmanuel Baccelli, and Adam Wolisz. 2016. "Industrial Wireless IP-Based Cyber Physical Systems." *Proceedings of the IEEE* 1025-1038.

Glossary

6TiSCH: IPv6 over the TSCH mode of IEEE802.15.4e 11

AES: Advanced Encryption Standard 12

CLI: Command Line Interface 21

CPU: Central Processing Unit 12

FAA: Federal Aviation Administration 22

GPIO: General-Purpose Input/Output 25

IC: Integrated Circuit 19

IEEE: Institute of Electrical and Electronics Engineers 9

IoT: Internet of Things 24

IPC: Inter-Processor Communication 15

LAKE: IETF Lightweight Authenticated Key Exchange 11

MAC: Medium Access Control 15

MEMS: Micro-Electromechanical System 18

OS: Operating System 22

RF: Radio Frequency 24

RTC: Real-Time Clock 18

RTCA: Radio Technical Commission for Aeronautics 22

RTOS: Real-Time Operating System 22

SCP: Secure Copy Protocol 45

TCXO: Temperature compensated crystal oscillator 18

TFTP: Trivial File Transfer Protocol 45

TSCH: Time-Slotted Channel Hopping 9

WSN: Wireless Sensor Networks 12