Pervasive and Mobile Computing 115 (2026) 102133

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

RTXBEE: Real-time communication module for critical Internet of Things applications

Valentin Stangaciu [a],*, Cristina Stangaciu [a], Daniel-Ioan Curiac [b], Mihai V. Micea [a]

[a] Department of Computer and Information Technology, Politehnica University Timisoara, 2, Vasile Parvan Blvd, Timisoara, 300006, Timis, Romania

[b] Automation and Applied Informatics, Politehnica University Timisoara, 2, Vasile Parvan Blvd, Timisoara, 300006, Timis, Romania

ARTICLE INFO

Keywords:
Real-time communication
Internet of Things
XBEE
Sensor networks

ABSTRACT

The Internet of Things concept has expanded to a large area of applications evolving to the point of providing even real-time support. Critical applications become increasingly suitable at the Edge Layer where real-time operations need to be supported at both node and network level thus communication becomes crucial. This paper presents a real-time communication solution based on the highly popular XBee modules. We describe a predictable and modular driver for such modules along with a full communication platform ready to be integrated into an IoT design for real-time applications. The proposed communication module has been implemented at prototype level and successfully validated through an extensive set of simulations and experiments.

1. Introduction

The development of real-time and critical applications in the Internet of Things (IoT) concept is only natural, therefore expanding the applicability of IoT to safety critical domains [1]. Being organized in three basic functional layers, namely Edge, Fog and Cloud [2], an IoT network is intended to provide time critical monitoring and control functionalities at Edge Layer where the implementation is generally represented by Wireless Sensor and Actuator Networks (WSANs) [3,4].

In order to adapt the Edge Layer of IoT to support critical applications, each software and hardware component of this layer must be able to function in a real-time manner [5]. Such a requirement must not only be applied at node level through specific application developing techniques and usage of a Real-Time Operating System (RTOS) [6] but must also be accompanied with real-time communication [7] between the nodes of the network where the infrastructure permits it.

An extremely popular communication solution used in IoT Edge Layer is represented by the ZigBee stack [8] with significant applicability in home automation or smart city [9], industrial applications [10], environmental monitoring and control systems [11,12], automotive [13] or healthcare [14,15]. Moreover, ZigBee's popularity and stability led to important studies and improvements conducted by the scientific community in terms of security [16] or performance [17].

There are many hardware RF modules available on the market that implement the ZigBee stack. Some of the available solutions are mature enough to be easily integrated into new IoT platforms but the lack of specifically designed drivers make this attempt questionable. Such a solution is represented by the XBee device family provided by Digi International. The XBee RF device family provides IoT solutions not only for ZigBee implementations [18] or their own proprietary similar solution DigiMesh [19] but also for Lora [20], Cellular LTE [21] and even lower level sub 1 GHz [22,23] interfaces based also on DigiMesh. Such devices are used in many new IoT or Wireless Sensor Networks (WSN) designs [24,25], are involved in scientifical studies [26–29] and are also used as teaching platforms [30], mainly because they are a mature and easy to integrate technology.

* Corresponding author.
E-mail address: valentin.stangaciu@cs.upt.ro (V. Stangaciu).

https://doi.org/10.1016/j.pmcj.2025.102133
Received 1 April 2025; Received in revised form 2 September 2025; Accepted 6 November 2025
Available online 7 November 2025

In this paper we present RTXBEE, a design and implementation for an embedded, platform independent real-time driver solution for the XBee family. This solution extends our previously conducted experiments presented in [31] where an attempt was made in this direction but with limited real-time functionality. We demonstrate the predictability of our solution with extensive experiments and measurements. Furthermore, we present a real-time hardware platform ready to be integrated into predictable real-time IoT designs. Our hardware solution fully implements the capabilities of the XBee module while providing a hard real-time full stack SPI communication interface, based on PARSECS_RT [32], directly available to be integrated into an IoT platform. Our main contributions presented in this paper are the following:

- A modular, platform independent real-time driver to manage the XBee family devices for critical systems.
- Based on our previous work, we extend the solution with additional software layers in order to provide a full real-time and predictable design.
- A double buffering solution was added in order to avoid any data loss from the XBee module.
- An integrated hardware module prototype ready to be integrated into a new IoT design, implementing RTXBEE along with our full stack hard real-time SPI communication protocol, namely PARSECS_RT.
- A full application layer API using PARSECS_RT as transport protocol.
- We evaluate our work on both simulations and a real hardware platform through extensive experiments.

The reminder of the paper is organized as follows. A related work study is presented in Section 2. Section 3 presents our design, coined as RTXBEE, while analyzing its real-time characteristics in Section 4. We provide the relevant experimental results in Section 5 whose aim is to support our solution, while concluding this paper in Section 6.

2. Related work

While XBEE devices have proven their efficiency by providing the communication interface in many projects from various fields, the software driver still poses an issue when targeting critical and industrial IoT applications.

Regarding the software driver solutions required to interface the XBee modules with embedded systems, the manufacturer offers a variety of implementations for many platforms and programming languages. A notable solution is the framework written in ANSI C [33] for many hardware architectures and operating systems. This solution has the best potential for adapting it for critical IoT application. The main advantage of Digi's ANSI C driver is that it offers an extremely detailed implementation for most of their devices. On the other hand, the driver does not offer any time guarantees although, in some platform ports, the implementation use non-blocking I/O operations. Furthermore, its potential for critical applications can also be justified that the driver's architecture is based on a main process function implementing the driver's relative tick which is called by the user. However, given its current limitations, it is not suitable for hard real-time applications.

Java [34] or Python [35] implementations from Digi are also available to be included into embedded applications. These solutions are generally designed as a library with detailed implementation for all the functionalities of the XBee family modules. However, these implementations, even if they are ported to some architectures, they are designed bu using a classical programming model using blocking procedures which do not provide execution predictability. This aspect is crucial for real-time systems thus the manufacture provided solution detrimentally affect their real-life application.

Aside from manufacturer provided drives, other third party solution offer similar features. An implementation close to the IoT paradigm is the one provided for the RIOT (Real-time Internet of Things Operating System) operating system [36] but it is limited to only certain XBee modules. Furthermore, even if RIOT targets IoT applications its real-time guarantees are extremely limited thus, in a best case scenario, it may be suited for soft real-time applications. The may reason behind this limitation is that it provides deterministic, preemptive, priority-based scheduling having even low latency interrupt support but without offering any deadline guarantees [37].

Another interesting approach is represented by a Linux kernel module implementing the XBEE driver [38]. This intriguing solution offers a very limited implementation for only one type of XBEE device through a network device serving as the interface with the user. Although its design is versatile the time constraints are not taken into account.

To the best of our knowledge, even with the continuous increase of the XBee family popularity, software solutions to support RF modules application in real-time scenarios are still in an infancy stage. The main disadvantage of the existing solutions is represented by the lack of predictability thus making their integration into real-time systems problematic.

In Table 1 we summarize the solutions we analyzed so far, by concentrating on the real-time aspects as well as their platform dependency. In a best case scenario, existing drivers may be used in soft real-time applications but, to the best of our knowledge, a hard real-time approach was not studied, leaving RTXBEE to tackle this gap for critical applications.

3. RTXBEE: XBEE real time driver

The work presented here is related to an initial experimental solution described in [31]. In this early work, only a partial idea was presented with limited real-time support and compatible with a deprecated version of an XBee modules [39]. This initial experimental solution introduced a simple three layer architecture for XBee Series 1 with limited functionalities and real-time capabilities, thus only the first two layers were able to achieve hard real-time constraints.

Table 1
Solution comparison.

Solution	Real-time aspects	Hardware/software platform	Programming Language
Digi XBee ANSI C Library [33]	Soft real-time	specific ports: DOS, mbed, Win32, Posix	C
Digi XBee Java Library [34]	No real-time	Hardware Platform Independent - needs Java runtime	Java
Digi XBee Python library [35]	No real-time	Python running hardware/software platform	Python
RIOT XBee Driver [36]	Soft real-time	RIOT Operating System	C
Rob ODwyer XBEE Kernel Module [38]	No real-time	Linux	C
RTXBEE	Hard Real-Time	Platform independent (except for Layer 1)	C

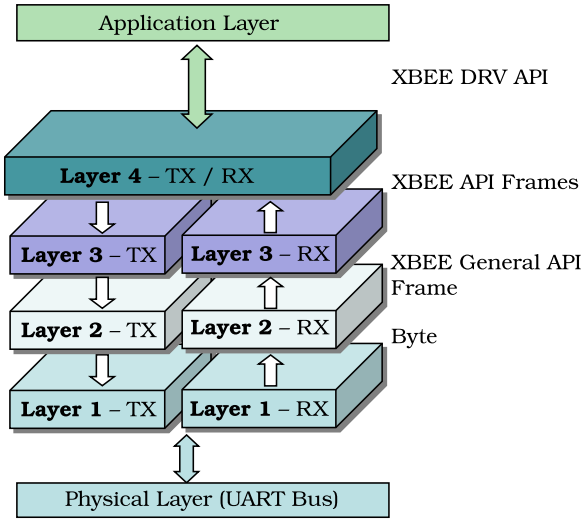

Fig. 1. RTXBEE Driver Architecture.

In this paper we present a full real-time up-to-date solution for the XBee family devices designed not only for the ZigBee compatible devices but also for other RF modules from the same device family. The solution presented here was tested and evaluated in many scenarios and enabled the design and development of many other related IoT projects.

It is important to begin with a crucial observation regarding the XBee device family: all of the RF modules available are compatible in terms of data transport and encoding thus the serialization is identical. Such an aspect greatly influenced our driver architecture making it easily adaptable for all the related RF modules.

The solution we provide is organized as a layered architecture as presented in Fig. 1. Each layer was designed to be implemented as a single task, but this does not limit the flexibility of implementation.

The first layer of RTXBEE is responsible to implement the basic UART communication with the XBee RF module at byte level. This layer is divided into two submodules, one for reception (*Layer 1 - RX*) and one for transmission (*Layer 1 - TX*). The interfacing between Layer 1 and the Layer 2 is represented by a pair of ring buffers, one for transmission and one for reception as described in Fig. 2.

As it can be observed in Fig. 2, the RX ring buffer, denominated as *xbee_rx_ring_buffer* is written by the reception (RX) component of Layer 1 when a byte is received and it is read by Layer 2. The same trivial functionality is handled by the transmission (TX) component of Layer 1 using the TX ring buffer, namely *xbee_tx_ring_buffer*. Although our solution was designed to be platform independent, this layer is clearly dependent on the hardware platform thus it needs to implement platform specific operation in order to handle the UART bus.

The raw bytes handled by Layer 1 are serialized and de-serialized by Layer 2 in order to obtain a *XBEE General API Frame*. This frame is identical for all the modules of the RF XBee family and it is used to transport module specific commands and data. The structure of the *XBEE General API Frame* is depicted in Fig. 3.

The frame begins with a Start-of-Frame (SOF) byte with the fixed value of 0x7E, followed by a two byte long length field (LEN) which states the length of the payload in the DATA field. The frame is concluded by a checksum field meant for error detection and

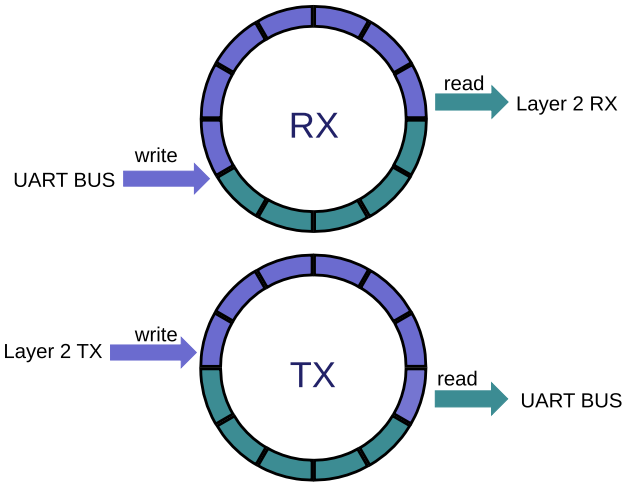

Fig. 2. Layer 1 interfacing with Layer 2 using ring buffers.

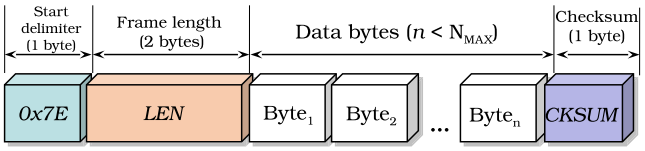

Fig. 3. XBEE General API Frame structure.

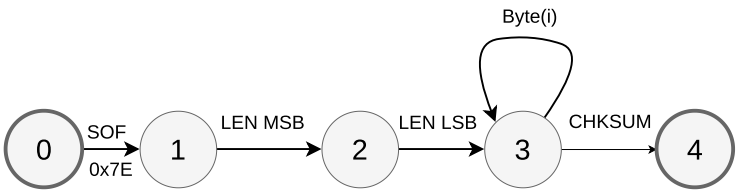

Fig. 4. XBEE General API Frame Assembly state machine.

calculated using (1) according to the manufacturer's official documentation.

$$CKSUM = 0xFF - \sum_{i=1}^{n} Byte_i \tag{1}$$

Layer 2 is also divided into two submodules one for RX and one for TX. Its main role is to assemble a correct and valid *XBEE General API Frame* from the raw bytes received from Layer 1 RX. The same principle is applied to the TX flow. The frame assembly process is low on complexity and its state machine is described in Fig. 4.

The *XBEE General API Frame* assembly process considers MSB reception byte order and is started upon the arrival of SOF byte (with the fixed value of 0x7E). The process continues on receiving the LEN field in state ① respectively ② after it continues to process the DATA field. The final role of this finite state machine is to receive the CHKSUM byte and then to validate the correctness of the received data.

The interfacing of Layer 2 with the upper layer (i.e. Layer 3) is done through a series of buffers as depicted in Fig. 5. It is noteworthy to mention that on the TX path only a single buffers provide the interfacing with the upper layers thus the XBee module cannot handle multiple *XBEE General API Frames* at the same time. On the other hand, the RX path is subjected to a double buffering interface. The reason behind this decision is to avoid any packet loss in the situation when the upper layers fail to process a received frame in time. Such a situation will not occur in the case when the full version of RTXBEE is used. However, we designed this solution in a modular way such as the layers may be used independently.

The two layers presented so far, namely Layer 1 and Layer 2, were designed to be compatible with any module in the XBee family thus, until this point, the interfacing is not module specific. In the following paragraphs we will present the rest of RTXBEE, represented by Layer 3 and Layer 4, which provide a full solution for the XBee modules implementing the ZigBee Stack, namely the XBee Series 3 [18].

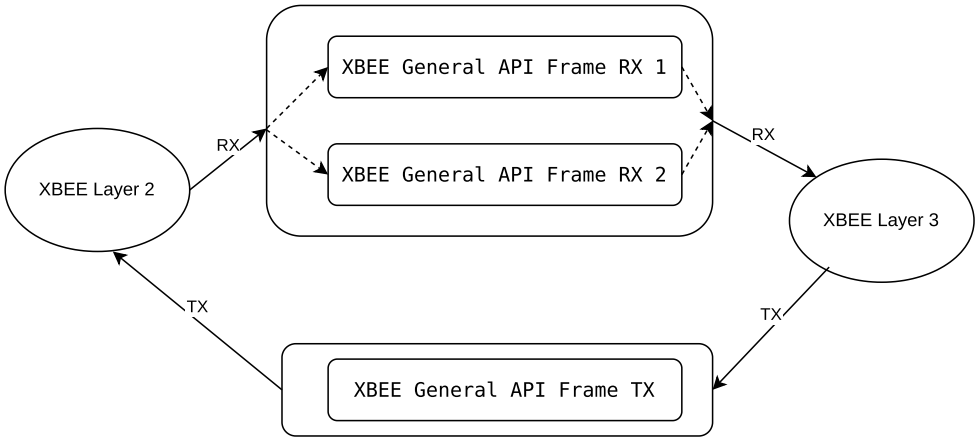

Fig. 5. Layer 2 interfacing with Layer 3 using single TX buffer and RX double buffering.

The main role of the Layer 3 of RTXBEE is to encode/decode specific API frames based on the unique identifier provided by the first data byte (i.e., $Byte_1$) of the *XBEE General API Frame* from Fig. 3. On the RX flow, this layer reads a *XBEE General API Frame* from one of the two buffers written by Layer 2, decodes and validates a correct XBEE RF Series 3 API frame, and finally stores the specific resulted frame into a message queue, denominated as *layer_3_rx_message_queue*, later to be processed by Layer 4.

On the other hand, the transmission processes has a similar flow: Layer 4 interfaces with Layer 3 through a transmission message queue, *layer_3_tx_message_queue*, which is consumed by Layer 3 in order to encode a *XBEE General API Frame* to be forwarded to the rest of the stack. It is important to mention that on either path, RX or TX, if the API frame is not valid, this layer will ignore the data, thus not forwarding the invalid frame to the upper/lower layer.

The most complex layer of RTXBEE is Layer 4 which is responsible for the entire control and monitoring of the XBee module. This layer's flow aims to provide a working and transparent RF ZigBee communication to the user in order to implement its application.

The flow of Layer 4, presented in Fig. 6, begins with a hardware reset procedure represented here by *XBEE Module Reset* state, involving a low pulse of 1 ms on the module's $\overline{RESET}$ pin. This ensures a predictable and clean start of the XBee module. Next, the flow waits a 5 s time period to allow the XBee module to fully power-up prior to initiating the module configure procedure designated as *XBEE Module Configure*. The configuration procedure is more complex and it will be detailed in the following paragraphs. As it can be observed, the *XBEE Module Configure* state may generate certain errors which may or may not be recoverable. Errors such as timeout or *XBEE Module Not Joined* will result in a restart of the main flow of Layer 4, while errors such as *MODULE NOT COMPATIBLE* or *MODULE FAULT* will block the entire flow of RTXBEE, such events being clearly not recoverable.

A successful module configuration will lead to a fully active XBee module in state *XBEE Module Idle and Ready* after transitioning through a preidle state, namely *XBEE Module PreIdle*. The module is fully usable by the application through dedicated APIs while remaining in *XBEE Module Idle and Ready*. This state is also responsible for keeping the XBee module in working conditions while detecting possible functional issues. In such situations, the flow will transition to the already presented error states as shown in Fig. 6.

Regarding the *XBEE Module PreIdle*, its purpose is to start periodic tasks that may occur such as a periodical node discovery before switching to the final ready state.

As discussed before, a critical state in the main flow of Layer 4 is represented by the *XBEE Module Configure* in Fig. 6. The role of this part of the flow is to identify the XBee module, configure it according to the parameters specified by the application layer and finally to ensure that the module has joined the ZigBee network thus making it active for communication.

The configuration flow, described in the diagram in Fig. 7, firstly identifies the module by reading its MAC address in the form of a 64 bit serial number. In the next step, the state of the XBee module's internal network join status is checked. This implies periodical polling the module for the network join status but subjected to a timeout. If the network association timeout expires, RTXBEE treats this event as a potential error thus triggering a fresh start of the main flow of Layer 4 beginning with a hardware reset of the module.

The network join operation is considered crucial, the XBee module being useless without taking part in a ZigBee network. After a successful network join, the configuration flow continues by interrogating the XBee module for certain parameters such as: operating channel, operating PAN ID, Network Address, Node Identifier, Power Level, Maximum Hops, Hardware and Software version. After collecting these parameters, the configuration flow concludes thus the main flow of Layer 4 considers the XBee module to be ready for the application layer.

As it can be observed in Fig. 7, the configuration flow can also signal errors to the main flow such as either recoverable ones as communication timeout or also unrecoverable errors such as *MODULE NOT COMPATIBLE* or *MODULE FAULT*.

According to the architecture diagram presented in Fig. 1, Layer 4 is the final upper layer of RTXBEE's stack, directly interfacing with the user's application layer. The main functions of the API available to the user are described in Listing 1.

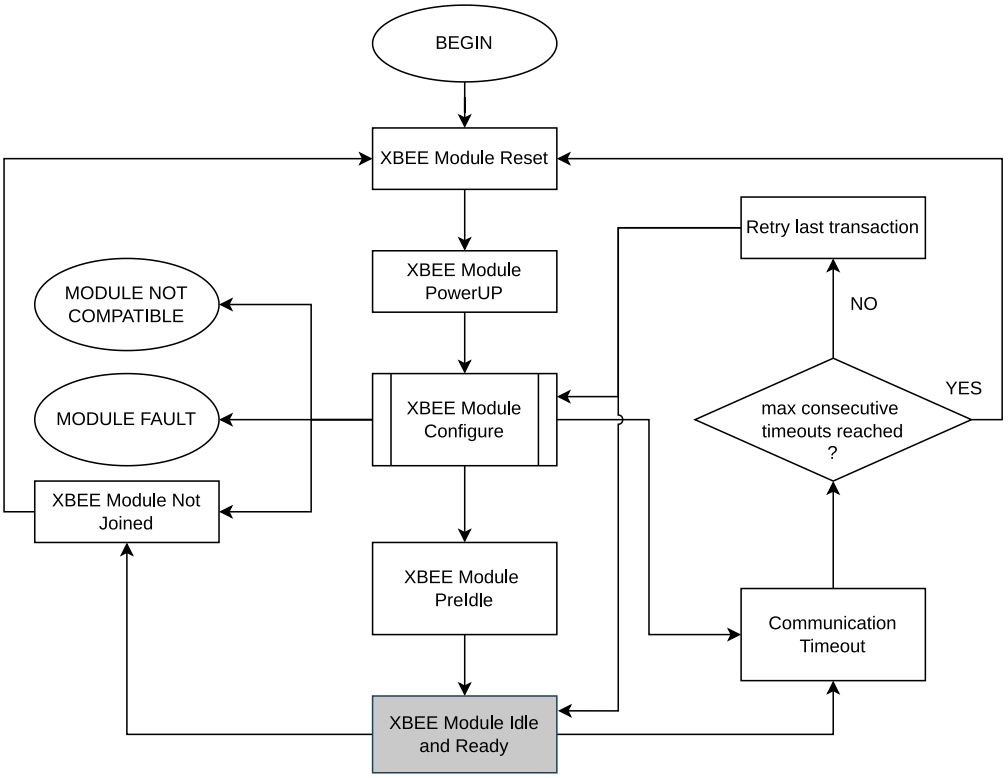

Fig. 6. Layer 4 Main Flow.

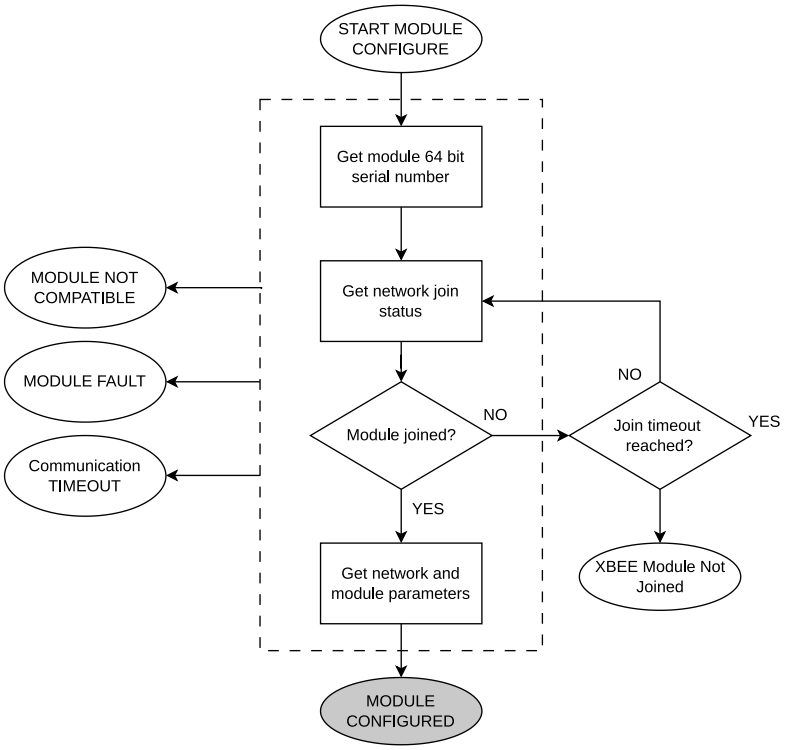

Fig. 7. Layer 4 XBee Series 3 Module Configure Flow.

```
XBEE2_MODULE_SETTINGS RTXBEE_GetSettings();
bool RTXBEE_SetSettings(RTXBEE_MODULE_SETTINGS *new_settings);
bool RTXBEE_FlushQueues();
RTXBEE_MODULE_STATE RTXBEE_GetState();
bool RTXBEE_RX_Packet(RTXBEE_RX_PACK *packet);
bool RTXBEE_TX_Packet(RTXBEE_TX_PACK *packet, uint8_t *sent_frame_id);
NODE_LIST RTXBEE_GetNodeList();
XBEE_STATISTICS GetStatistics();
```

Listing 1: RTXBEE User API

The *RTXBEE_GetSettings* and *RTXBEE_SetSettings* provide an interface for the user to read or change the main settings of the XBee module. The call of the *RTXBEE_SetSettings* API will inevitably trigger the restart of the main flow of Layer 4. The settings of the XBee module are implemented using the structure presented in Listing 2.

```
typedef struct
{
  uint8_t CH_OperatingChannel;
  uint64_t OP_OperatingPanID;
  uint16_t ID_PANID;
  uint16_t MY_NetworkAddress;
  uint64_t SHSL_SerialNumber;
  char* NI_NodeIdentifier;
  POWER_LEVEL PL_PowerLevel;
  uint16_t VR_FirmwareVersion;
  uint16_t HV_HardwareVersion;
  XBEE2_ASSOCIATION_INDICATION_STATUS AI_AssociationIndication;
  uint8_t NH_MaximumUnicastHops;
  uint64_t unicast_transmission_timeout;
}XBEE2_MODULE_SETTINGS;
```

Listing 2: XBee Module Settings Structure

Using RTXBEE API, the user may read the settings in Listing 2 through *RTXBEE_GetSettings* or may also configure settings through *RTXBEE_SetSettings* such as the node identifier, power level or maximum unicast hops.

```
typedef enum
{
  XBEE2_STATE_BEGIN,
  XBEE2_STATE_RESET,
  XBEE2_STATE_POWERUP,
  XBEE2_STATE_CONFIGURE,
  XBEE2_STATE_PREIDLE,
  XBEE2_STATE_IDLE,
  XBEE2_STATE_MODULE_NOT_RESPONDING,
  XBEE2_STATE_MODULE_NOT_RESPONDING_RETRY,
  XBEE2_STATE_MODULE_NOT_COMPATIBLE,
  XBEE2_STATE_MODULE_NOT_JOINED,
  XBEE2_STATE_MODULE_NOT_JOINED_RETRY,
  XBEE2_STATE_MODULE_FAULT,
}XBEE2_MODULE_STATE;
```

Listing 3: RTXBEE Driver State

The current state of the RTXBEE driver, as described in Listing 3, may be queried by the user through the *RTXBEE_GetState* function, while the list of the other nodes in the network, obtained using the node discovery feature, may be interrogated through the *NODE_LIST RTXBEE_GetNodeList* API.

The user may interrogate the RTXBEE driver for an already received RF packet through *RTXBEE_RT_Packet*. If an RF packet was received, this call will return *true* and will write the content in the *packet* parameter. In order to sent an RF packet the user will use the *RTXBEE_TX_Packet* providing the packet to be sent through the *packet* parameter. In case of success, this call will return *true* while writing the XBEE frame_id into the output parameter *send_frame_id*.

Additionally, RTXBEE provides a statistical feedback to the user using the *GetStatistics* API from Listing 1. The statistical information available to the user is presented in Listing 4:

```
typedef struct
{
  uint32_t hardware_tx_retries;
  uint32_t rx_packet_count;
  uint32_t tx_packet_count;
  uint32_t tx_packet_no_ack_count;
  uint32_t tx_packet_cca_fail_count;
```

```
  uint32_t tx_packet_purged_count;
  uint32_t tx_packet_invalid_dest_point_count;
  uint32_t tx_packet_net_ack_fail_count;
  uint32_t tx_packet_not_joined_count;
  uint32_t tx_packet_self_addr_count;
  uint32_t tx_packet_addr_not_found_count;
  uint32_t tx_packet_no_route_count;
}XBEE_STATISTICS;
```

Listing 4: RTXBEE Driver RF Module Statistical data

Such statistical information may be of interest to the user to either debug certain network communication issues, evaluate network performance or even identify a possible faulty XBee RF module.

4. Real-time analysis

The main reason for designing RTXBEE was to provide a predictable hard real-time solution for the popular XBee modules in order to be applied into time critical IoT networks. This work is not intended to offer any guarantees regarding the functionality of the XBee module itself but we will demonstrate the real-time behavior of the RTXBEE driver. In order to be able to function in a real-time manner, RTXBEE requires the support of an underlying Real Time Operating System (RTOS) such as FreeRTOS [40], a Linux system with real-time extensions similar to Litmus^{RT} [41] or using latest Linux extension sched_ext [42]. Furthermore, even if proper real-time support is offered by the RTOS, the task timings still need to be properly calculated.

This section will provide a formal analysis regarding the real-time aspects of RTXBEE. In order to begin this analysis we will need to make the following simple considerations: (i) each Layer of RTXBEE as described in Fig. 1 will be implemented as a single task, (ii) the processor used to implement RTXBEE may have a hardware receive FIFO within its UART controller, (iii) the application layer, not being part of RTXBEE will not be analyzed but some minimum requirements about its time constraints will be overviewed.

We will first define the minimum execution frequency for the task implementing Layer 1 of RTXBEE, denoted as F_{L1}. Considering that in order to effectively transfer a single byte over the UART interface at a transfer speed identified by $BAUD$ measured in bps (bits per second), a number of 10 bits is required (8 data bits, 1 START bit and 1 STOP bit) and using $FIFO$ to identify the number of bytes of the hardware FIFO of the UART controller (a value of $FIFO = 1$ meaning no FIFO is available), the minimum execution frequency of the task implementing Layer 1, $F_{MIN}(L1)$, can be calculated using:

$$F_{MIN}(L1)[Hz] = \frac{BAUD[bps]}{10 \cdot FIFO} \tag{2}$$

Considering (2), the frequency of the task implementing Layer 1 needs to satisfy the following condition:

$$F_{L1} \geq F_{MIN}(L1) \tag{3}$$

Usually, in RTOSs time restrictions are given as task execution periods rather than frequencies. In order to ease the rest of our analysis we will consider:

- T_{L1} - the execution period of the task implementing Layer 1 where $T_{L1} = \frac{1}{F_{L1}}$
- T_{L2} - the execution period of the task implementing Layer 2
- T_{L3} - the execution period of the task implementing Layer 3
- T_{L4} - the execution period of the task implementing Layer 4

According to the official documentation [18], the smallest API frame is the Modem Status frame having only the mandatory Frame Type along with one byte of payload. By adding the encoding of the *XBEE General API Frame* as defined in Fig. 3 we can calculate that the minimum size of a *XBEE General API Frame* is 6 bytes. This value will impose the maximum period of the execution of the task implementing Layer 2 (T_{L2}):

$$T_{L2} \leq 6 \cdot T_{L1} \tag{4}$$

Regarding Layer 3, its execution period is dictated by the double buffering technique used to provide the interfacing with Layer 2:

$$T_{L3} \leq 2 \cdot T_{L2} \tag{5}$$

As for Layer 4, its execution period is imposed by the size of the reception and transmission queues, denoted as $QUEUE\_SIZE$ that handle the interfacing with Layer 3:

$$T_{L4} \leq QUEUE\_SIZE \cdot T_{L3} \tag{6}$$

All of the above calculations provide the time conditions that need to be satisfied in order to provide a correct functionality, otherwise, RTXBEE will clearly experience packet loss. The task execution periods are best handled when an RTOS is used. However, on a practical approach, the schedulability is highly influenced by the amount of total tasks running in the system as well as the Worst Case Execution Time (WCET) of each task. The later is clearly dependent on the CPU that executes these tasks.

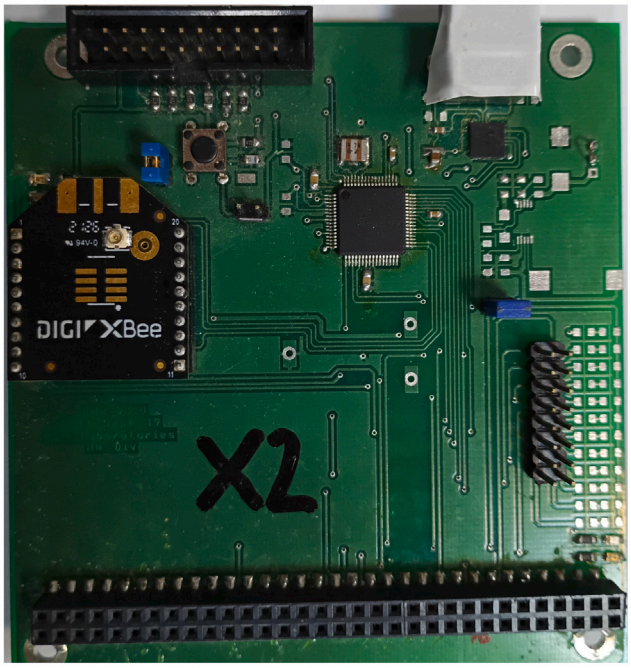

Fig. 8. Real-Time communication module for XBee.

5. Implementation and experimental results

We implemented the RTXBEE solution mainly on hardware platforms that include autonomous communication modules ready to be integrated into IoT nodes in order to provide a predictable communication.

One such implementation is a communication board for a real-time modular multiprocessor IoT node. This communication module, as presented in Fig. 8, has its own microcontroller unit (MCU) which on one side manages the XBee RF module using our RTXBEE solution and, on the other hand, offers an API through an SPI interface using the PARSECS_RT hard real-time communication stack [32].

The API support of PARSECS_RT provides the means to get the value of a parameter, set the value of a parameter or call a remote procedure. The RTXBEE parameters that may be requested through the PARSECS_RT stack are presented in Abstract Syntax Notation One (ASN.1) Standard [43] in Listing 5.

```
PARAMETER_TYPE_ID ::= ENUMERATED {
XBEE_MY_NETWORK_ADDRESS                 (0x01),
XBEE_SN_SERIAL_NUMBER                   (0x02),
XBEE_OP_OPERATING_PANID                 (0x03),
XBEE_ID_PANID                           (0x04),
XBEE_CH_OPERATING_CHANNEL               (0x05),
XBEE_VR_FIRMWARE_VERSION                (0x06),
XBEE_HV_HARDWARE_VERSION                (0x07),
XBEE_PL_POWER_LEVEL                     (0x08),
XBEE_ND_COUNT                           (0x09),
XBEE_ND_NODE_INDEX                      (0x0A),
XBEE_TX_TIMEOUT_VALUE                   (0x0B),
XBEE_STATISTICS                         (0x0C),
RTXBEE_VERSION                          (0xFE),
}
```

Listing 5: PARSECS_RT API - RTXBEE GetRequest Parameter List

As described in Section 3, some of the RF module's parameters may be written by the user, thus finally causing a reconfiguration procedure in order to apply the new settings. These parameters are exported through the PARSECS_RT API as enumerated in Listing 6.

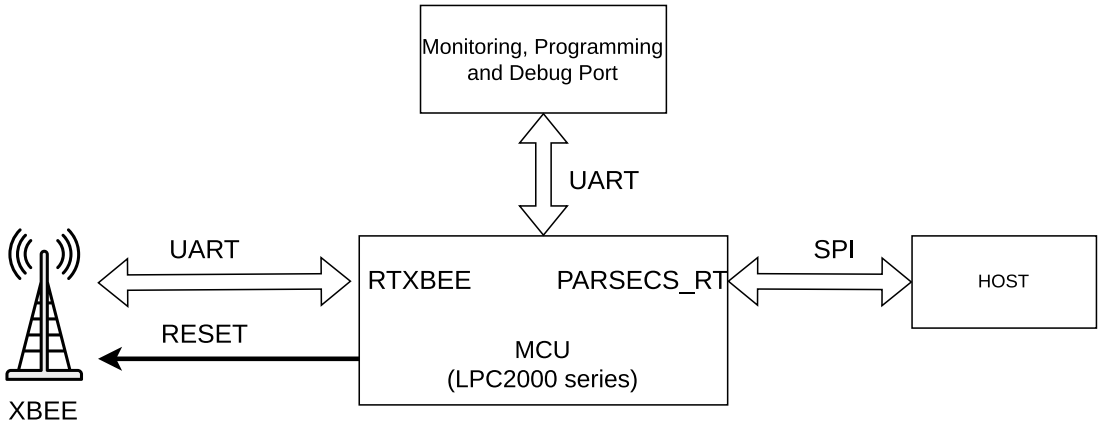

Fig. 9. Real-Time communication module for XBee - Block schematic.

```
PARAMETER_TYPE_ID ::= ENUMERATED {
XBEE_PL_POWER_LEVEL                    (0x08),
XBEE_TX_TIMEOUT_VALUE                  (0x0B),
RTXBEE_VERSION                         (0xFE),
}
```

Listing 6: PARSECS_RT API - RTXBEE SetRequest Parameter List

The packet exchange is implemented using the CallRequest API provided by PARSECS_RT with the methods enumerated in Listing 7.

```
METHOD_TYPE_ID ::= ENUMERATED {
XBEE_RECV_PACKET_NO_CONFIRM                    (0x01),
XBEE_SEND_PACKET_NO_CONFIRM                    (0x14),
XBEE_SEND_PACKET_WITH_CONFIRM                  (0x15),
}
```

Listing 7: PARSECS_RT API - RTXBEE CallRequest Method List

The user may request to send an RF packet using a remote CallRequest API identified by either *XBEE_SEND_PACKET_NO_CONFIRM* or by using the API identified by *XBEE_SEND_PACKET_WITH_CONFIRM*. On the other hand, on the reception of a new RF packet from the network, the host, connected via SPI to our real-time communication module, will be notified by another remote CallRequest identified by the method type *XBEE_RECV_PACKET_NO_CONFIRM*.

As stated before, the real-time communication module presented in Fig. 8, uses RTXBEE to handle the XBee RF module and PARSECS_RT for the real time SPI communication. Our hardware is built around a microcontroller manufactured by NXP, the LPC2000 [44] family, as depicted in Fig. 9. The LPC2000 microcontrollers are based on the ARM7TDMI-S architecture known for their predictable execution. The LPC2000 processor family are used in many embedded projects having a large amount of peripheral devices include the UART and SPI which are significant for our solution [45]. The MCU is connected to the XBEE device via one of its UART interfaces along with the dedicated GPIO pin in order to provide the hardware reset. On the other hand, the MCU also interfaces with the host platform via the SPI bus which is handled by the PARSECS_RT communication protocol transporting the above presented API.

The operating system of our real-time communication module is represented by FreeRTOS [40]. The need for such an RTOS is crucial in order to achieve the real-time constraints and have a deadline guarantee.

A detailed description of the most important parameters regarding the hardware and operating system performance is provided in Table 2.

All of the timings required for certain operations within the RTXBEE driver are handled by real-time software timers [46,47] having the resolution fixed to 1 ms. In order to implement all the required timings, six software timers were employed:

- **xbee_powerup_timer** - used to implement timings for hardware reset and module power up. According to the hardware specifications of the XBEE module the reset low pulse was set to 300 ms. After the reset the module will need an additional period of 7000 ms to fully boot.
- **xbee_general_timeout_timer** - used to implement all of the communication timeout situations. The duration of this timeout is variable and it depends on each operation.

Table 2
Real time communication module parameters.

Parameter	Value
CPU Core Freq	58.9824 MHz
CPU Peripheral Freq	14.7456 MHz
CPU RAM Memory	32 KB
CPU Flash Memory	128 KB
XBee BAUD Rate	57 600 bps
FreeRTOS Version	V11.1.0
FreeRTOS Heap Size	12 KB
FreeRTOS Tick	10 KHz

Table 3
Communication Module Running Tasks.

Task name	Task short name	Task description
Task_TimerSoftware	TS	The software timer engine that allows the RTXBEE timings to be implemented
Task_XBeeLayer1	XBEE1	Layer 1 RTXBEE
Task_XBeeLayer2	XBEE2	Layer 2 RTXBEE
Task_XBeeLayer3	XBEE3	Layer 3 RTXBEE
Task_XBeeLayer4	XBEE4	Layer 4 RTXBEE
Task_PARSECS_LL	SPI_LL	PARSECS_RT Low Level Substack
Task_PARSECS_HL	SPI_HL	PARSECS_RT High Level Substack
Task_AppTask_1	APP1	Internal Application task 1
Task_AppTask_2	APP2	Internal Application task 2

- **xbee_configuration_timeout_timer** - used to implement the configuration procedure timeout. This timer practically guards the configuration process and limits this procedure to a value of 10 s. According to the driver flows presented earlier this timeout will trigger a full reconfiguration process of the module starting with a full reset procedure.
- **xbee_not_responding_retry_timer** - used to implement a general timeout that triggers a module not responding event which leads to a restart of the Layer 4 Main Flow in Fig. 6
- **xbee_node_discovery_period_timer** - a periodic timer that initiates a node discovery procedure. This timer keeps track of the periodic node discovery process where the XBee module practically triggers a network scan. The period for this procedure is configurable and set as default to 60 s
- **xbee_association_timer** - used to implement the network join timeout. This timer is crucial to the configuration process thus it practically defines the node's association timeout. The purpose of the XBee node is to be associated to the network. If the node fails to correctly associate to an existing network this timer will trigger a full reconfiguration process beginning with a hardware reset.

The real-time communication module presented in Fig. 8 offers a complete predictable communication solution using 9 FreeRTOS tasks as described in Table 3 while their interaction flow is depicted in Fig. 10.

As it can be observed in Table 3 and in Fig. 10 the main tasks responsible for implementing RTXBEE are denominated as XBEE1-4. Also, in order to obtain the timings required by the driver, software timers were used as presented earlier in this chapter, which are handled by the TimerSoftware task designated as TS. The PARSECS_RT communication stack is achieved by the SPI_LL and SPI_HL responsible for the Low Level and the High Level Substack. The last two tasks, denominated as APP1 and APP2, are responsible for implementing the main flow of the application of the communication module presented in Fig. 8 . The role of the APP1 task is to forward all the RF data packets from the XBee network to the master host connected via PARSECS_RT SPI protocol with our communication module. On the hand, a similar role is attributed to the APP2 task. This task must mainly handle the API over the PARSECS_RT protocol as described in Listing 5, Listing 6 and Listing 7.

The interaction and the execution flow of these tasks is depicted in Fig. 10. The purpose of the communication module prototype that we designed as the board in Fig. 8 having the hardware diagram in Fig. 9 and the task flow in Fig. 10 is to provide an off-the-shelf communication module ready to be integrated into a new IoT design where the time constraints represent crucial requirements.

In order for these tasks to be properly executed by a RTOS, in our case FreeRTOS, certain time parameters need to be determined, such as the task execution period along with its WCET.

In Table 4 we present the time parameters for each task, along with the execution priorities.

The execution period for the tasks implementing the RTXBEE driver (i.e. XBEE1-4) were calculated using Eqs. (2), (3), (4), (5) and (6) considering the values in Table 2 focusing on the XBee BAUD rate of 57600 bps.

An accurate calculation of the WCET still poses a significant challenge, thus we obtained this parameter experimentally through extensive measurements with the help of a logic analyzer. Some detailed examples regarding this aspect will be presented later in this section. Using the same principle, along with the WCET values, we determined the Average Execution Time (AET) and the Minimum Execution Time (MET).

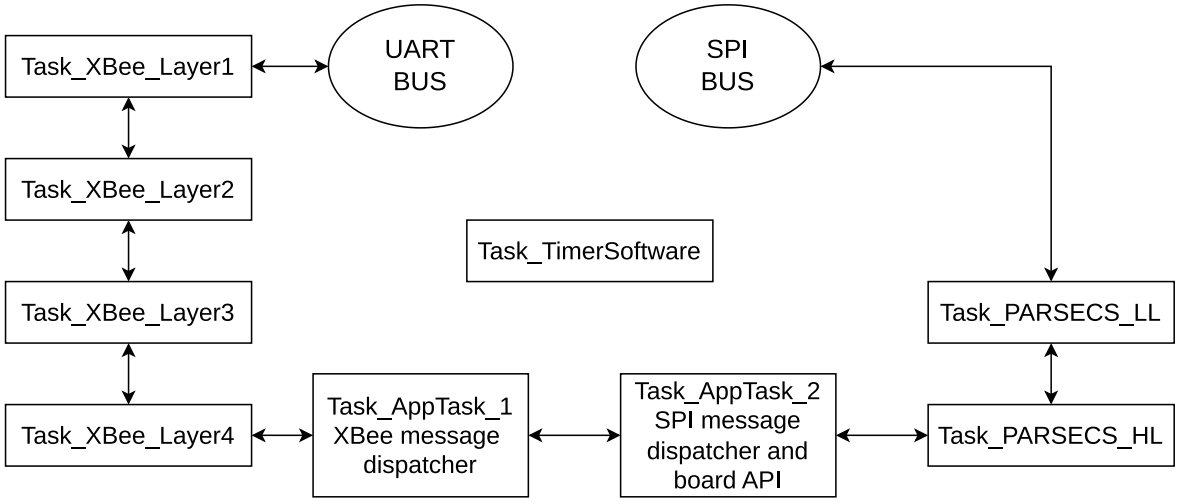

Fig. 10. Real-Time communication module for XBee - Task communication flow.

Table 4
Task time parameters.

Task	Criticality priority	Rate mon. priority	Period [μs]	WCET [μs]	AET [μs]	MET [μs]
TS	9	8	1000	40	23	17
XBEE1	8	7	2500	150	6	5.5
XBEE2	7	4	15 000	240	5.8	3.5
XBEE3	5	3	30 000	250	10	9.5
XBEE4	5	2	50 000	2800	59.5	3
SPI_LL	8	9	500	50	15	10.5
SPI_HL	6	6	3000	450	9.9	9.5
APP1	5	1	50 000	2100	26	1.5
APP2	5	5	12 500	1800	8	3

Along with these values, in Table 4 we added the task priorities that we used in our experiments. As FreeRTOS, like many real-time operating systems, provides a priority based scheduling, we need to determine and specify the priority for each task. We considered two possible approaches: give priorities based on tasks criticality (i.e. importance, in this context) or based on their activation frequency (i.e. Rate Monotonic priority).

In our implementation, we have chosen the latter, because by doing so, we actually use the Rate Monotonic scheduling algorithm [48,49], a static priority based algorithm in which the priorities are assigned based on the task activation rate (i.e. the task with the highest activation frequency, implicitly with the lowest period, is assigned the highest priority, the next priority is assigned to the task with the next activation rate value, and so on). This algorithm is known to be optimal among uniprocessor fixed priority scheduling algorithms [48]. Regarding the tasks periods, we have chosen to harmonically relate their values with respect to the smallest value. As it can be observed in Table 4, the task having the minimum period is SPI_LL, thus, we have chosen all of the other task periods to be a multiple of the period of the SPI_LL task, while still respecting the time constrains imposed for each task by the Eqs. (2), (3), (4), (5) and (6). We have chosen this approach in order to increase the general Rate Monotonic utilization bound for processor utilization which is about 0.69 (i.e. 69%) by default. By using proper harmonic relations between periods, this bound can be theoretically raised to 100% [50].

Another way of assigning priorities is to take into consideration the task's level of importance in the system, using the direct priority scheduling approach. In our scenario, FreeRTOS was configured to support 10 levels of priorities, from 0 to 9, where 9 is the highest priority level. The priorities were assigned from the most important level to the lesser ones. Thus, TS task was assigned the highest priority as the application directly depends on it, because it provides the time support for functionalities implemented by tasks, e.g. timeout. The XBEE1 and SPI_LL received the next lower priority level, and so on.

Another crucial aspect that needs to analyzed is regarded to CPU usage in three different situations: worst case execution, average execution and minimum execution (idle). While the idle or average CPU usage is useful in most cases, in real-time environments the absolute maximum CPU usage needs to be considered.

$$U[\%] = \frac{ET}{T} \cdot 100 \tag{7}$$

Table 5
CPU Usage.

Task	MAX CPU %	AVG CPU %	MIN CPU %
TS	4.00	2.30	1.70
XBEE1	6.00	0.24	0.22
XBEE2	1.60	0.04	0.02
XBEE3	0.83	0.03	0.03
XBEE4	5.60	0.12	0.01
SPI_LL	10.00	3.00	2.10
SPI_HL	15.00	0.33	0.32
APP1	4.20	0.05	0.01
APP2	14.40	0.06	0.02
TOTAL	**61.63**	**6.18**	**4.43**

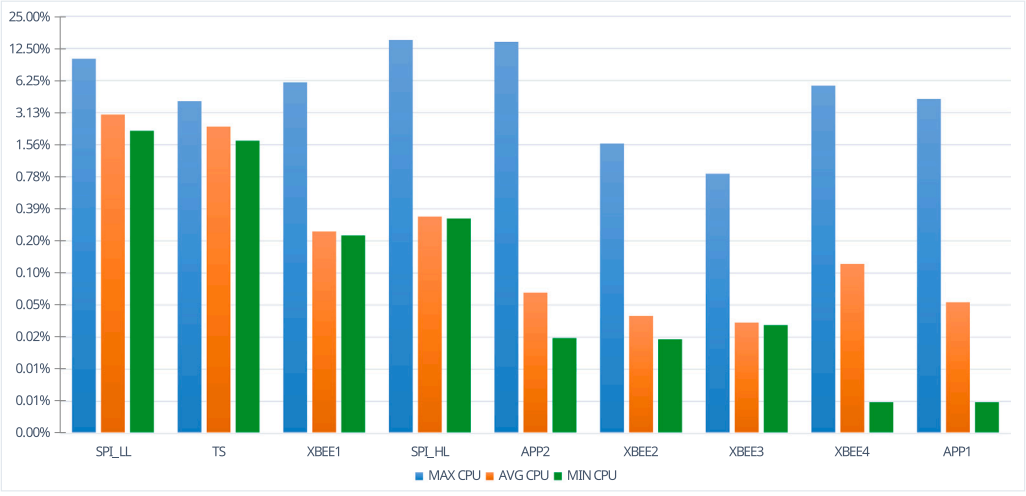

Fig. 11. CPU Usage.

In order to calculate the execution time for each task in any scenario we make use of the equation in (7) where ET represents the execution time of the task and T the task execution period. Using the above Eq. (7) over the values in Table 4 we obtained the maximum, average and minimum CPU usage values presented in Table 5.

As it can be observed, the total CPU usage in a worst case scenario is presenting as a reasonable value thus still allowing other tasks to be added in order to extend current functionalities. On the other hand, comparing this value with the average and minimum CPU usage we can clearly point out the noticeable difference suggesting a very pessimistic behavior for the worst case scenario as it is true for most of the real-time systems. Because the maximum CPU usage value is below 69%, the task set is schedulable with Rate Monotonic algorithm regardless of the task period values. Still, in order to increase the schedulability up to 100% it is recommended to use harmonic period values.

In order to ease the analysis of the CPU usage, Fig. 11 depicts the values from Table 5 in a graphical manner for each task using a base 2 logarithmic axis for CPU usage. This representation helps us understand the pessimistic estimation for each task and also to have a clear overview over the total CPU usage of the system. We can easily observe that in the case of tasks SPI_LL and TS the maximum CPU usage based on WCET is not too pessimistic as we would expect, thus, in an average usage these two tasks need roughly the same amount of CPU TIME. The situation is different for the rest of the tasks where the CPU is not actually fully used in an average execution. Clearly, overall, the system will allow the execution of very low priority tasks such as logging, debugging or monitoring tasks given the pessimistic estimation.

The important question at hand is why the WCET estimation is so pessimistic in the cast of most tasks. As we stated before in this paper, the WCET parameter was determined experimentally by measuring the execution time of each task in the situation where the workload was maximum. The WCET estimation was measured for each task individually. Considering this aspect, in a normal usage of the system, it is extremely unlikely that in a moment in time all the tasks will occupy the processor as in a WCET estimation. However, in a real-time scenario, the system needs to be ready for such a situation.

We validated the task set schedulability using values from Table 2 in the SimSo simulator [51] for a hyperperiod of 150000 μs, value given by the least common multiple of the tasks' periods. The first part of the simulation is depicted in Figs. 12 and 13, where a time unit represents 10 μs. The simulations were performed using both Rate Monotonic and Criticality based approach for assigning priorities and both proved approached proved to be feasible.

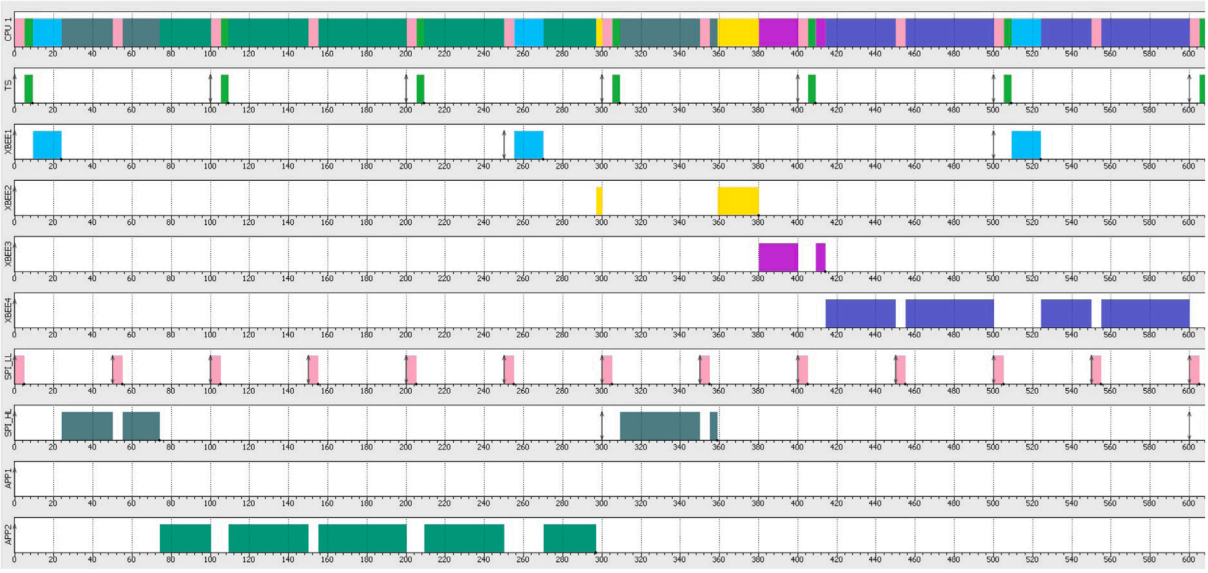

Fig. 12. Task Simulation in SimSo, using WCET execution model and Rate Monotonic Priorities.

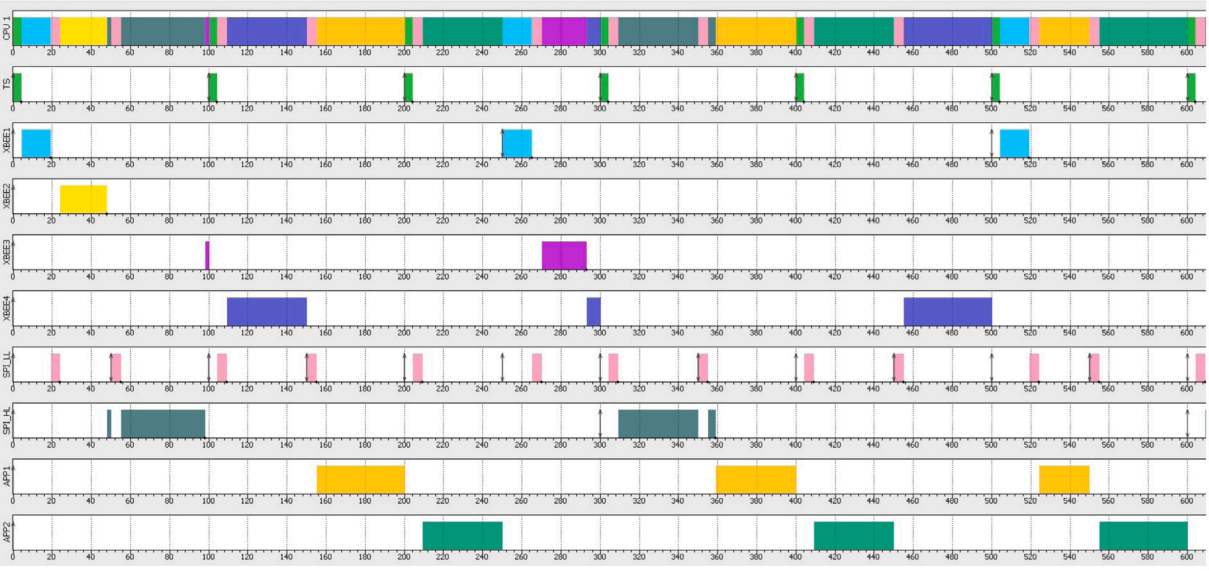

Fig. 13. Task Simulation in SimSo, using WCET execution model and Criticality-based Priorities.

As we can see from Figs. 12 and 13, there is no deadline miss in either of the cases. Thus for this CPU utilization level both approaches for assigning priorities depicted in Table 4 are feasible. The difference between them resides only in the order of the task execution.

Considering the fact that we designed this driver for embedded systems low on resources, the analysis of the memory footprint is necessary. With the aid of memory analysis APIs provided by FreeRTOS, we were able to extract the memory usage of the software modules present in the system as presented in Table 6 reaching a total footprint of about 23 KB.

In order to further demonstrate and analyze our solution in term of real-time behavior, we performed measurements of the tasks timings using a Saleae Logic Pro16 Analyzer [52]. In order to obtain the time measurements we used classical embedded approach: we used a dedicated GPIO for each task which is toggled to logic LOW at the beginning of the task execution and back to logic HIGH at end of the task execution. The waveforms were then captured using the Logic Analyzer. A sample of these measurements is presented in Fig. 14.

Table 6
Memory Usage.

Software Component	FreeRTOS Stack [bytes]	Data memory [bytes]	Total memory [bytes]
TimerSoftware	96	402	498
RTXBEE	1344	7333	8677
PARSECS_RT	888	6071	6959
APP	1144	3807	4951
CPU Stack		512	512
FreeRTOS		996	996
TOTAL			**22593**

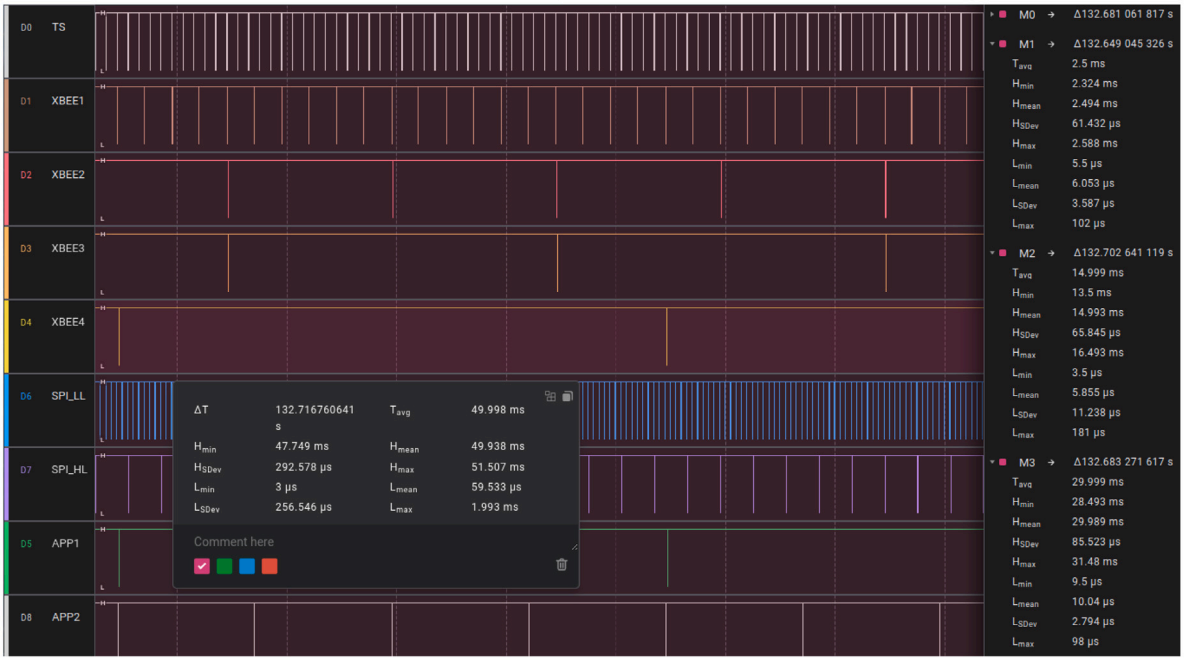

Fig. 14. Logic Analyzer Task Timings Measurements.

To better explain the presented measurements we used timing interval markers for each channel. Fig. 14 depicts the measurements for all the tasks described in Table 3 but we concentrated only on the tasks regarding our RTXBEE solution. For the first three tasks of RTXBEE (XBEE1, XBEE2, XBEE3) the time measurements are presented on the right side of the figure designated as M1, M2 and M3. The timing measurements for the last task of RTXBEE, namely XBEE4, are presented in the floating window on the bottom of the figure. We used the values of L_{MAX}, L_{AVG} and L_{MIN} to measure the WCET, AET and MET while the task period may be identified by the T_{AVG} field.

Given the fact that we designed RTXBEE to be platform independent (except for Layer 1) we could easily adapt it in order to evaluate its functionalities and deterministic behavior on another hardware platform and operating system. RTXBEE was thus ported on a Raspberry PI3 Model B V1.2 having a quad core CPU running at 1.2 GHz with 1 GB of RAM memory [53]. The XBee module was connected to the Raspberry PI3 board via a simple USB to Serial Adapter. The operating system is ArchLinuxARM having the Litmus^{RT} [41] kernel patch applied. We took advantage of the 4 CPU cores and we configured the Litmus^{RT} scheduler having the parameters as described in 7.

A similar execution context as the one for the LPC2000 implementation is presented in the logic analyzer capture in Fig. 15, having the same time parameters measured as in Fig. 14.

While considering the space limitations to present the extended data in the paper, we also provided the captured measurements as supplementary materials to this paper [54]. The data repository contains 2 saved captures from the Logical Analyzer which can be studied by using the Saleae Logic software [52]. Furthermore, in order to reproduce these measurements two extensions need to be installed along the main Logic Analyzer software, directly from the Extensions Manager: Pulse Stats by Peter Jaquiery [55] and Pulse Interval Stats by Coriander V. Pines [56]. It is also important to mention that all the timing markers have been included in the supplementary capture files.

Table 7
Task time periods - RTXBEE on Linux with Litmus.

Task	Litmus priority	CPU Affinity	Litmus Reservation ID	Period μs
TS	2	1	1002	1000
XBEE1	1	2	1003	1200
XBEE2	2	2	1004	6000
XBEE3	3	3	1005	10 000
XBEE4	4	3	1006	40 000
SPI_LL	3	0	1000	1000
SPI_HL	5	0	1001	6000

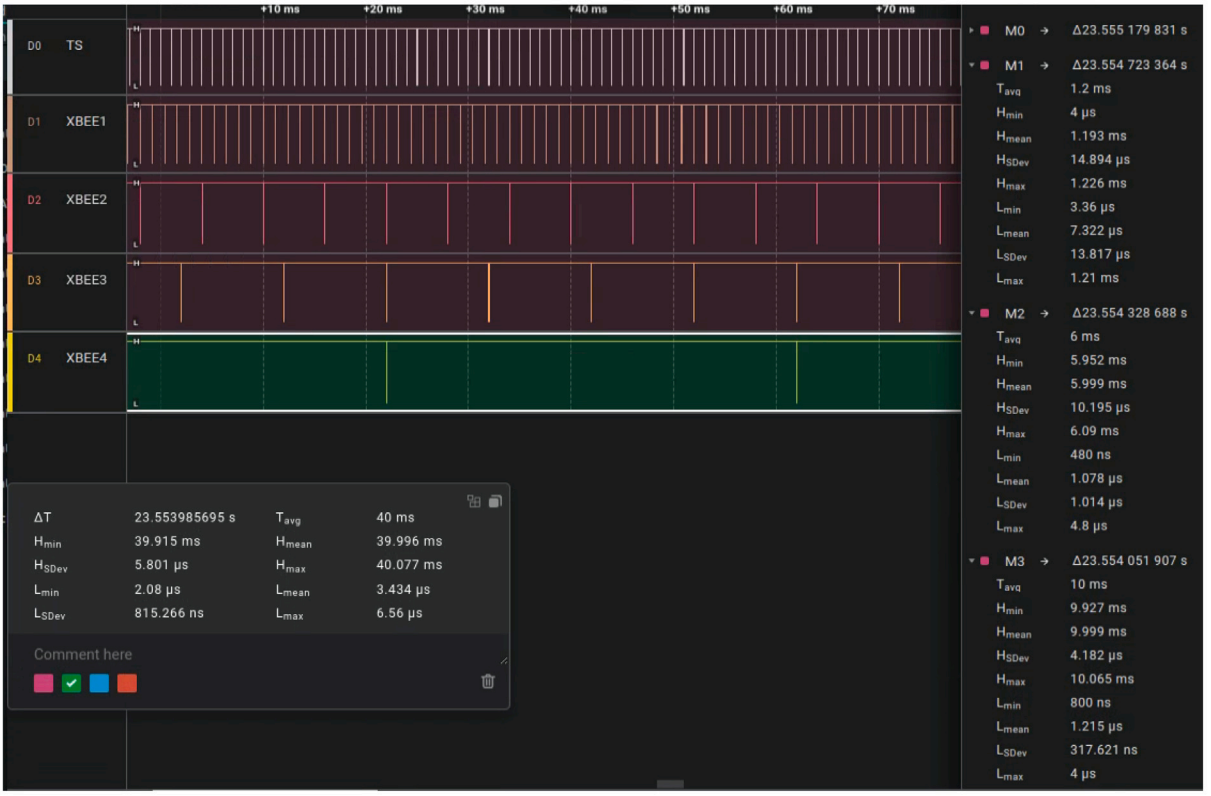

Fig. 15. Logic Analyzer Task Timings Measurements - RTXBEE on Litmus.

The key aspect of our solution is represented by its predictability, stability and determinism compared to a classical approach. In order to prove the stability of RTXBEE we compared its execution against the manufacturer provided solution [33]. Considering the same evaluation setup with the Raspberry PI3 platform, along with the same task context, we evaluated RTXBEE and Digi-XBEE ANSI C library from the deterministic point of view.

The determinism was evaluated by performing a set of measurements of the same transaction (a request and a response) with the XBEE module. We have chosen a simple transaction where we request the least significant part of the serial number by issuing the SL command to the XBEE module and reading back its response. The justification for this choice is that we need to establish the determinism of the RTXBEE stack compared to the manufacturer driver and not the XBEE module itself. For instance, choosing an RF transmission transaction would not be appropriate thus the transaction period would be greatly influenced by the RF communication medium.

The measurement result of the transaction period using RTXBEE and the manufacturer provided driver is presented in Fig. 16 where each point represents a transaction period. The red plot represents the execution periods of the manufacturer provided driver while the blue plot represented the transaction period of RTXBEE.

We can clearly observe that even if the performance of RTXBEE is lower compared to the manufacturer's driver, the predictability and stability of RTXBEE is easily observed. In a real-time system determinism is the key factor having a greater importance than performance.

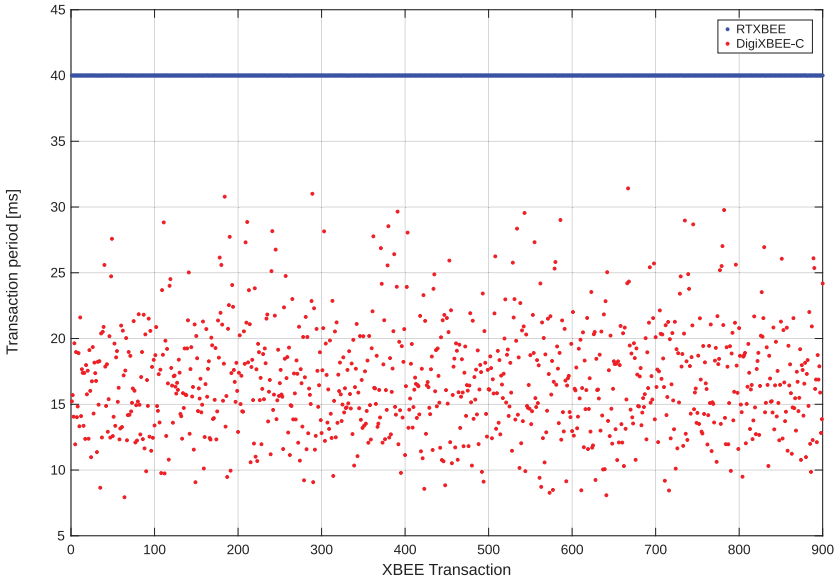

Fig. 16. XBEE Transaction Period.

6. Conclusions

In this paper we present RTXBEE, an embedded and real-time communication solution for IoT, based on the XBee RF modules. The architecture encompasses 4 layers where the first two layers provide a universal time-bounded implementation for the general API interface of the XBee modules family.

The latter two layers are dependent on the specific module that is used for communication. We channeled the functionality flow for the last two layers to fully implement the feature of the XBee Series 3 RF module oriented on ZigBee communication. The predictable methodology that we used in this case can be adapted to implement the functionalities of other types of RF modules from the XBee family.

Furthermore, a hardware prototype was built not only to demonstrate the effectiveness of RTXBEE but to also deliver an integrated communication solution to be integrated into an IoT node design. By integrating with the PARSECS_RT stack over SPI, we provided an API in order for a host connected to our prototype to be able to use XBee series RF modules for communicating within the Edge of an IoT network.

The real-time behavior was not only formally described but it was also validated through simulations and real hardware implementation having FreeRTOS as an underlying embedded real-time operating system.

In terms of scalability, the first two layers of RTXBEE may be used without any changes with all types of XBee devices currently available. Regarding the last two layers of RTXBEE, they need to be adapted for each device type specifically using the same methodology. Providing a version of RTXBEE for all the XBee family devices is currently our ongoing future work.

The current limitations of the work we present in this paper are represented by the fact that we implemented the functionalities for the XBee Series 3 for ZigBee networks. Although the first two layers of RTXBEE are universal for all the XBee device family, we intend to include all the RF modules into RTXBEE making it a full communication software solution for IoT.

RTXBEE has successfully proven its effectiveness in designing a hybrid platform where classical real-time wireless sensor networks were integrated into an IoT network by making use of the MQTT-SN application protocol [57] which was transported using our solution presented in this paper.

CRediT authorship contribution statement

Valentin Stangaciu: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Cristina Stangaciu:** Writing – review & editing, Validation, Software, Conceptualization. **Daniel-Ioan Curiac:** Writing – review & editing, Supervision, Formal analysis. **Mihai V. Micea:** Writing – review & editing, Supervision, Project administration, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] N.M. Quy, L.A. Ngoc, N.T. Ban, N.V. Hau, V.K. Quy, Edge computing for real-time internet of things applications: Future internet revolution, Wirel. Pers. Commun. 132 (2) (2023) 1423–1452, http://dx.doi.org/10.1007/s11277-023-10669-w.

[2] F. Firouzi, B. Farahani, A. Marinšek, The convergence and interplay of edge, fog, and cloud in the AI-driven internet of things (IoT), Inf. Syst. 107 (2022) 101840, http://dx.doi.org/10.1016/j.is.2021.101840.

[3] Y. Wang, S. Wu, C. Lei, J. Jiao, Q. Zhang, A review on wireless networked control system: The communication perspective, IEEE Internet Things J. 11 (5) (2024) 7499–7524, http://dx.doi.org/10.1109/jiot.2023.3342032.

[4] C.-L. Fok, G.-C. Roman, C. Lu, Adaptive service provisioning for enhanced energy efficiency and flexibility in wireless sensor networks, Sci. Comput. Program. 78 (2) (2013) 195–217, http://dx.doi.org/10.1016/j.scico.2011.12.006.

[5] J. Zhang, M. Ma, P. Wang, X.-d. Sun, Middleware for the internet of things: A survey on requirements, enabling technologies, and solutions, J. Syst. Archit. 117 (2021) 102098, http://dx.doi.org/10.1016/j.sysarc.2021.102098.

[6] S.-I. Hahm, J. Kim, A. Jeong, H. Yi, S. Chang, S.N. Kishore, A. Chauhan, S.P. Cherian, Reliable real-time operating system for IoT devices, IEEE Internet Things J. 8 (5) (2021) 3705–3716, http://dx.doi.org/10.1109/jiot.2020.3025612.

[7] P. Dong, Z. Jiang, A. Burns, Y. Ding, J. Ma, Build real-time communication for hybrid dual-OS system, J. Syst. Archit. 107 (2020) 101774, http://dx.doi.org/10.1016/j.sysarc.2020.101774.

[8] A. Zohourian, S. Dadkhah, E.C.P. Neto, H. Mahdikhani, P.K. Danso, H. Molyneaux, A.A. Ghorbani, IoT Zigbee device security: A comprehensive review, Internet Things 22 (2023) 100791, http://dx.doi.org/10.1016/j.iot.2023.100791.

[9] Z. Lv, B. Hu, H. Lv, Infrastructure monitoring and operation for smart cities based on IoT system, IEEE Trans. Ind. Inform. 16 (3) (2020) 1957–1962, http://dx.doi.org/10.1109/tii.2019.2913535.

[10] S. Ding, J. Liu, M. Yue, The use of ZigBee wireless communication technology in industrial automation control, Wirel. Commun. Mob. Comput. 2021 (1) (2021) http://dx.doi.org/10.1155/2021/8317862.

[11] S.-C. Hu, Y.-C. Wang, C.-Y. Huang, Y.-C. Tseng, Measuring air quality in city areas by vehicular wireless sensor networks, J. Syst. Softw. 84 (11) (2011) 2005–2012, http://dx.doi.org/10.1016/j.jss.2011.06.043.

[12] G. Zhang, X. Liu, F. Zheng, Y. Sun, G. Liu, Geological disaster information sharing based on internet of things standardization, Environ. Earth Sci. 83 (5) (2024) http://dx.doi.org/10.1007/s12665-023-11353-9.

[13] A. Palaniappan, R. Muthiah, M. Tiruchi Sundaram, ZigBee enabled IoT based intelligent lane control system for autonomous agricultural electric vehicle application, SoftwareX 23 (2023) 101512, http://dx.doi.org/10.1016/j.softx.2023.101512.

[14] S. Chen, H. Zhi, H. Zhang, J. Wang, X. Li, Application of integrated medical care "cloud-based virtual ward" management model on postoperative analgesia: Based on zigbee technology, Pain Manag. Nurs. 26 (1) (2025) 23–29, http://dx.doi.org/10.1016/j.pmn.2024.07.011.

[15] J. Qi, P. Yang, G. Min, O. Amft, F. Dong, L. Xu, Advanced internet of things for personalised healthcare systems: A survey, Pervasive Mob. Comput. 41 (2017) 132–149, http://dx.doi.org/10.1016/j.pmcj.2017.06.018.

[16] A. Zohourian, S. Dadkhah, E.C.P. Neto, H. Mahdikhani, P.K. Danso, H. Molyneaux, A.A. Ghorbani, IoT Zigbee device security: A comprehensive review, Internet Things 22 (2023) 100791, http://dx.doi.org/10.1016/j.iot.2023.100791.

[17] B. Padma, S.B. Erukala, End-to-end communication protocol in IoT-enabled ZigBee network: Investigation and performance analysis, Internet Things 22 (2023) 100796, http://dx.doi.org/10.1016/j.iot.2023.100796.

[18] Digi International Inc., Digi XBee® 3 zigbee® RF module, 2023.

[19] Digi International Inc., Digi XBee® RR DigiMesh 2.4 RF module user guide, 2022.

[20] Digi International, Digi XBee LR RF Module for LoRaWAN, URL https://www.digi.com/products/embedded-systems/digi-x-on/digi-xbee-lr-for-lorawan.

[21] Digi International Inc., Digi XBee® 3 cellular LTE-M/NB-IoT global smart modem user guide, 2024.

[22] Digi International Inc., Digi XBee® XR 868 RF module user guide, 2024.

[23] Digi International Inc., Digi XBee® XR 900 RF module user guide, 2024.

[24] P. Fernández-Bustamante, I. Calvo, E. Villar, O. Barambones, Centralized MPPT based on sliding mode control and XBee 900 MHz for PV systems, Int. J. Electr. Power Energy Syst. 153 (2023) 109350, http://dx.doi.org/10.1016/j.ijepes.2023.109350.

[25] I. Calvo, J.M. Gil-García, E. Villar, A. Fernández, J. Velasco, O. Barambones, C. Napole, P. Fernández-Bustamante, Design and performance of a xbee 900 MHz acquisition system aimed at industrial applications, Appl. Sci. 11 (17) (2021) 8174, http://dx.doi.org/10.3390/app11178174.

[26] V. Gavra, O.A. Pop, I. Dobra, A comprehensive analysis: Evaluating security characteristics of xbee devices against zigbee protocol, Sensors 23 (21) (2023) 8736, http://dx.doi.org/10.3390/s23218736.

[27] K.F. Haque, A. Abdelgawad, K. Yelamarthi, Comprehensive performance analysis of zigbee communication: An experimental approach with XBee S2C module, Sensors 22 (9) (2022) 3245, http://dx.doi.org/10.3390/s22093245.

[28] M. Orgon, L. Zagajec, I. Schmidt, Xbee technology: Complex evaluation of radio parameters, in: 2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops, ICUMT, IEEE, 2019, pp. 1–6, http://dx.doi.org/10.1109/icumt48472.2019.8970753.

[29] N. Hiron, A. Andang, N. Busaeri, Investigation of wireless communication from under seawater to open air with Xbee pro S2B based on IEEE 802.15.4 (case study: West java pangandaran offshore Indonesia), in: Proceedings of the Future Technologies Conference, FTC 2018, Springer International Publishing, 2018, pp. 672–681, http://dx.doi.org/10.1007/978-3-030-02683-7_47.

[30] C. Bell, Beginning Sensor Networks with Xbee, Raspberry Pi, and Arduino: sensing the World with Python and Micropython, A Press, 2020, http://dx.doi.org/10.1007/978-1-4842-5796-8.

[31] M.V. Micea, V. Stangaciu, C. Stangaciu, C. Filote, Sensor-level real-time support for xbee-based wireless communication, in: Proceedings of the 2011 2nd International Congress on Computer Applications and Computational Science, Springer, 2012, pp. 147–154.

[32] V. Stangaciu, C. Stangaciu, D.-I. Curiac, M.V. Micea, PARSECS_RT: A real-time PARSECS-based communication protocol stack for critical sensing applications, Internet Things 25 (2024) 101139, http://dx.doi.org/10.1016/j.iot.2024.101139.

[33] Digi International Inc., Digi XBee ANSI c library, 2024, https://github.com/digidotcom/xbee_ansic_library.

[34] Digi International Inc., XBee java library, 2024, https://github.com/digidotcom/xbee-java.

[35] Digi International Inc., Digi xbee python library, 2024, https://github.com/digidotcom/xbee-python.

[36] RIOT, RIOT XBee Driver, URL https://doc.riot-os.org/group_drivers_xbee.html.

[37] B. Patel, P. Shah, Operating system support, protocol stack with key concerns and testbed facilities for IoT: A case study perspective, J. King Saud Univ. - Comput. Inf. Sci. 34 (8) (2022) 5420–5434, http://dx.doi.org/10.1016/j.jksuci.2021.01.002.

[38] R. ODwyer, XBEE driver, 2009, https://github.com/robbles/xbee-driver.

[39] MaxStream, Inc., XBee/XBee-PRO OEM RF modules, 2007.

[40] R. Barry, FreeRTOS Reference Manual: API Functions and Configuration Options, Real Time Engineers Limited, 2009.
[41] R. Spliet, M. Vanga, B.B. Brandenburg, S. Dziadek, Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUSRT, in: 2014 IEEE Real-Time Systems Symposium, 2014, pp. 96–105, http://dx.doi.org/10.1109/RTSS.2014.33.
[42] The Linux Kernel development community, Extensible scheduler class, 2025, https://www.kernel.org/doc/html/next/scheduler/sched-ext.html. (Accessed 18 February 2025).
[43] International Telecommunication Union, Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), 2002.
[44] NXP Semiconductors, User Manual, Tech. Rep., Koninklijke Philips Electronics N.V, 2004.
[45] T.P. Martin, The Insider's Guide to the Philips ARM7-Based Microcontrollers an Engineer's Introduction to the LPC2100 Series, Hitex, 2005, p. 197.
[46] V. Stangaciu, C. Stangaciu, D. Curiac, Timer software: A software timer library for embedded real-time systems, 2023, http://dx.doi.org/10.2139/ssrn.4527250, Available At SSRN 4527250, URL https://ssrn.com/abstract=4527250.
[47] V. Stangaciu, TIMER SOFTWARE, 2023, https://github.com/svalentin1984/timer-software.
[48] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973) 46–61.
[49] N. Fisher, S. Baruah, Rate-monotonic scheduling, in: M.-Y. Kao (Ed.), Encyclopedia of Algorithms, Springer New York, New York, NY, 2016, pp. 1784–1788, http://dx.doi.org/10.1007/978-1-4939-2864-4_334.
[50] M. Mohaqeqi, M. Nasri, Y. Xu, A. Cervin, K.-E. Årzén, On the problem of finding optimal harmonic periods, in: Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 171–180, http://dx.doi.org/10.1145/2997465.2997490.
[51] SimSo - Simulation of Multiprocessor Scheduling with Overheads, 2023, https://projects.laas.fr/simso/. (Accessed 03 March 2023).
[52] Saleae Logic Analyzers, 2025, https://www.saleae.com. (Accessed 08 March 2025).
[53] Raspberry PI, Datasheet Raspberry PI 3 Model B Technical Specification. [Online], http://www.farnell.com/datasheets/2027912.pdf.
[54] V. Stangaciu, RTXBEE Supplementary Materials, https://vision.cs.upt.ro/s/JQi8HLzKHfBBNXW.
[55] P. Jaquiery, Pulse stats, 2021, https://github.com/GrandFatherADI/PulseStats.
[56] C.V. Pines, Pulse interval stats, 2022, https://github.com/cvpines/Pulse-Interval-Stats—Saleae-Logic-2.
[57] V. Stangaciu, C. Stangaciu, B. Gusita, D.-I. Curiac, Integrating real-time wireless sensor networks into IoT using MQTT-SN, J. Netw. Syst. Manage. 33 (2) (2025) http://dx.doi.org/10.1007/s10922-025-09916-1.