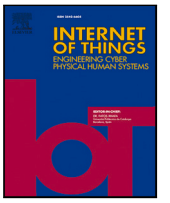
Internet of Things 25 (2024) 101139

Contents lists available at ScienceDirect

Internet of Things

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



Research article

PARSECS_RT: A real-time PARSECS-based communication protocol stack for critical sensing 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
Protocol stack
SPI bus
PARSECS protocol

ABSTRACT

The real-time characteristics of modular systems have been a long sought-after goal in many industries including automotive, aeronautics or mobile robotics. Yet scientists and practitioners are still struggling to find widespread implementation solutions that may accommodate diverse inter-modular real-time communication requirements. In an attempt to cover this research gap, the current paper proposes a real-time version of the PARSECS protocol for low-end devices. Our new protocol, coined as PARSECS_RT, was designed according to Open Systems Interconnection Reference Model. It offers a full communication stack from the physical layer to the application layer on top of the SPI interface in order to provide a stable, hard real-time communication platform. PARSECS_RT was evaluated in real and simulated environments providing promising results.

1. Introduction

Real-Time systems are ubiquitous nowadays in several areas of everyday life, especially in critical domains like process automation, avionics or automotive. In the last decade, even if the attention of the researchers in the field has been primarily focusing on transferring the real-time principles to new technologies, including Internet of Things (IoT) and Edge Computing, the problems encountered on end devices still remain unsolved. However, relatively new concepts as Real-Time Function-as-a-Service (RT-FaaS) [1] or Tactile Internet [2] are advancing the field of IoT by proposing real-time functionalities at superior levels, while taking the real-time functionality of the edge level for granted.

In this context, the complexity of the end devices, especially in the case of highly modular systems with multiple control and processing units [3], has increased tremendously in recent years. This trend is driven by the fact that, on one hand, these devices are constrained in terms of resources, performance, energy consumption and costs, and, on the other hand, being situated on the cyber–physical layer, they have more and more complex components for environment sensing, control and data processing. In these circumstances, assuring real-time characteristics for end devices becomes even more challenging. For example, in complex IoT systems, in order to comply with real-time requirements in a deterministic manner, one needs to provide real-time functionalities considering a bottom-up approach, that starts with the lowest level of the network, namely the end devices placed on the edge level and going up through different levels of IoT network (e.g., fog, cloud).

Thus, in order for a node of the network to respect time constraints all the components of the node need to function in a timely manner. A classical solution for providing real-time behavior at the node level is given by real-time operating systems such as FreeRTOS, RIOT [4] or HARETICK [5]. Nevertheless, the communication between its components also needs to respect

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

https://doi.org/10.1016/j.iot.2024.101139
Received 3 November 2023; Received in revised form 31 January 2024; Accepted 21 February 2024

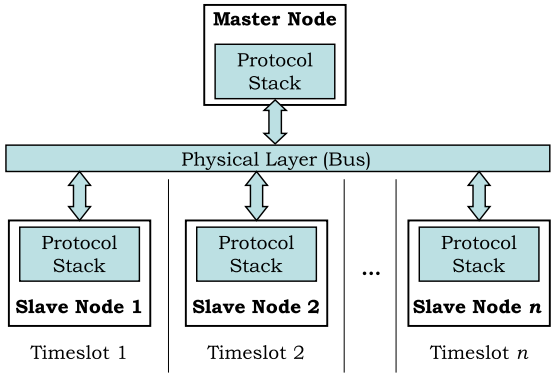


Fig. 1. PARSECS general connection architecture.

real-time constraints. Such a requirement can only be achieved by using specially designed communication protocols over the existing hardware buses. Such protocols need not only to provide a real-time communication stack but to also be lightweight adding encapsulation protocol data as less as possible making them able to be executed on architectures with low processing resources.

While, there is a consistent number of real-time communication protocol stacks for inter-node communication designed especially for wireless sensor networks [6–8] and IoT platforms [9–11], the things are significantly different regarding intra-node communication. Some research endeavours in this direction worth mentioning are based on 1-Wire Protocol, providing MicroLAN communication stack. Being almost a complete OSI stack, MicroLAN is ideal for sensor communication but not suitable between processing units, thus it is designed for low-power and low bandwidth communications for dedicated sensors [12]. Other stack level solutions such as CAN [13] for FlexRay [14] are not suitable for our scenario, as they are dedicated for the automotive industry and have fewer hardware implementation in microcontrollers.

In contrast, being an efficient, fast and synchronous communication platform, the SPI bus can be a good base for intra-node communication, from our perspective. As a consequence, our solution defines a real-time protocol stack in accordance with the OSI Reference Model, having the SPI bus as a communication base. Considering that the SPI bus enforces hard real-time constraints by design, our solution builds, upon it, a complete communication stack between processing units inside a system, while also providing a simple application layer API to the end user.

In this paper, we target a solution for communication, only at the node level, between different components located physically on the same node. The solution is based on the original design of PARSECS but adds multiple functionalities and optimizations. Our main contributions presented in this paper are the following:

- An adaptation of the PARSECS initial protocol for low-end devices, by using the OSI Reference Model;
- The protocol was divided into two separated and independent sub-stacks with a complete layer separation;
- The data packet of the Physical Layer was adapted by adding a packet type field and a sequence number, while the CRC polynomial was changed according to the RFC1662 specifications;
- A double buffering technique was added to the Data Link Layer in order to ensure no data loss;
- New layers were added: a transport layer in order to provide a reliable link between nodes in a TP0 CLNS profile, a presentation layer to provide data representation using Basic Encoding Rules and an application layer to provide a clear and structured API;
- An application test case is presented and analyzed from a real-time perspective.

The rest of the paper is organized as follows. Section 2 provides a brief description of the original PARSECS protocol. In Section 3, our new protocol, coined as PARSECS_RT, is presented in detail, while Sections 4 and 5 are devoted to real-time analysis from both theoretical and experimental points of view. Finally, conclusions are drawn in Section 6.

2. PARSECS protocol

In this section, we will discuss the first version of the protocol which was initially presented as PDCI (Predictable Data Communication Interface for hard Real-Time Systems) [15], PARSECS (Predictable Architecture for Sensor Communication Systems). It was designed to be a time-triggered hard real-time communication protocol over a full-duplex SPI bus [16]. The main goal is to provide flexible TDMA (Time Division Multiple Access) medium access control (MAC) communication for Hard Real-Time (HRT) systems over the SPI (Serial Peripheral Interface) bus based on a master–slave communication paradigm architecture as shown in Fig. 1. This initial version mainly lacked the superior levels of the stack and it did not provide a superior API, data encapsulation and segmentation.

PDCI is designed with a two layered architecture over of hardware SPI bus with an additional application layer providing the upper lever APIs. The two layers operate in a hard real-time environment and rely on an HRT operating system in order to accomplish

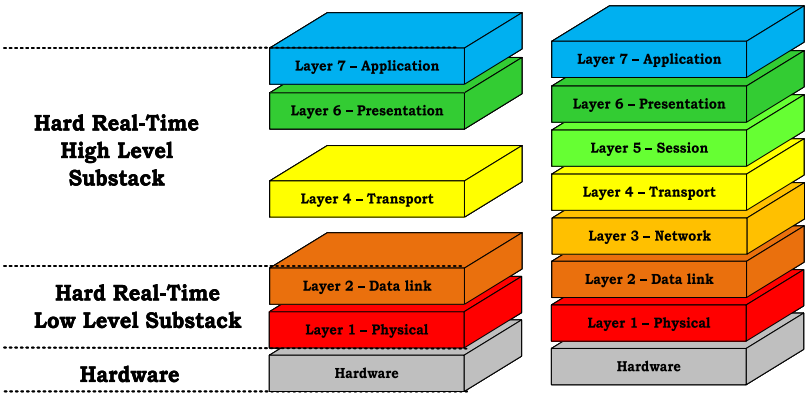


Fig. 2. (a) PARSECS_RT stack layers; (b) OSI Reference Model layers.

the time constraints. The first layer provides a physical interface control and handles both transmitted and received data at a byte level. The main role of this layer is to provide the necessary hardware abstraction layer to the upper layers of communication. The data is passed to a second layer which assembles the raw bytes in message frames, guarded by SOF (Start of Frame) and EOF (End of frame) control bytes thus obtaining a message-based data handling [15]. Such an approach, referred to later as PARSECS-M [16], requires certain word escaping procedures in order to avoid misinterpreting data as false SOF and EOF control bytes. On the other hand, the PARSECS-P approach [16] implements the second communication layer with packet-based frames insuring greater stability and error detection.

PARSECS later added a third layer on top of the modified PDCI in order to provide synchronization and data acknowledgment. Another final and upper layer, layer 4, is added only to provide the necessary APIs for the application. This latter layer does not require any real-time support.

The currently available implementation of PARSECS provides only the basic operations for HRT communication over the SPI bus, but it is not a full-stack solution and it does not offer compliance with the OSI reference model [17]. An improved solution is introduced in this paper, a new version of the PARSECS protocol, denominated as PARSECS_RT which offers a full stack HRT communication over the SPI bus.

3. PARSECS_RT protocol stack

In this paper, we propose an improved and more complex version of the PARSECS initial protocol, called PARSECS_RT, which is dedicated for the edge level of the IoT. In contrast with its predecessor, the PARSECS_RT protocol stack has a layered organization in full agreement with the rules of the OSI Reference Model, except for some unneeded layers which are not applicable in this type of communication. The OSI layers represent one of the most important references in IoT architectures [18]. The correspondence between these layers and the ones of the PARSECS_RT stack [17] is depicted in Fig. 2.

The network layer is currently not supported mainly because the layer's main role, i.e. routing, is not necessary for the architecture described above, thus SPI communication is based on the master–slave paradigm which does not require such a feature. The data flow is either from the master to one of the slaves or from a slave to the master, thus, no routing is needed. The other missing layer, layer 5 — Session, is also not needed in this architecture because of the obvious reason that on the SPI bus, all the communicating partners are always connected and the SPI bus is not dynamic in terms of clients. Adding or removing clients may involve physical changes in the electronic schematic of the module implementing the master of the communication. In the current case, where this layer is not supported, one may consider that for each client there is a session activated indefinitely.

In Fig. 1 the PARSECS_RT stack is presented in terms of layers. The hardware layer is practically not part of the stack thus it represents the actual hardware implementation of the bus. The PARSECS_RT stack is divided into two sub-stacks, both providing predictable and hard real-time operation.

The PARSECS_RT Low Level Sub-stack implements the lower layers of the PARSECS_RT SPI communication stack and provides a maximum of layer 2 capabilities according to the OSI Reference Model. It thus provides a predictable, hard real-time SPI communication ensuring stable and reliable communication between two directly connected nodes via the SPI bus with error detection and packet acknowledging. It contains the first two layers of PARSECS_RT: Layer 1 — Physical and Layer 2 — Data link. This sub-stack was designed to be implemented either as a single atomic task, as two single atomic tasks (one for Layer 1 and one for Layer 2), but also as three single atomic tasks (one task for Layer 1, one task for Layer 2 RX and another for Layer 2 TX). In this paper, we consider the sub-stack implemented as a single atomic task.

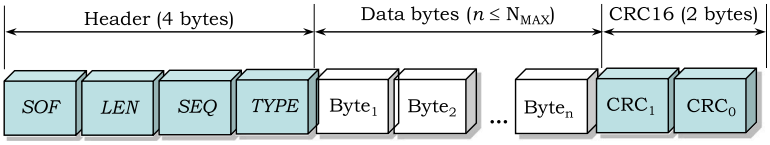


Fig. 3. SPI Base Frame structure.

3.1. Layer 1 - Physical Layer

The first part of the sub-stack is represented by Layer 1 — Physical Layer which is in total accordance with the OSI stack. Its main roles are to implement the hardware platform driver interface and provide basic, unstructured, raw data transfer on the SPI bus. This layer's implementation must be different from one hardware platform to another and also between master and slave devices, thus actually providing the hardware abstraction layer. This layer has only one component which incorporates both transmission and reception, thus in a full-duplex SPI communication such is the normal functioning behavior. Layer 1 specification and implementation must be different in terms of master and slave nodes. On the slave nodes, the implementation of the Layer 1 is quite trivial thus it only needs to implement the basic send–receive procedures as the hardware SPI peripheral practically does the rest. On the other hand, on the master node, the implementation of the Layer 1 is slightly different, thus it must poll each slave that is connected to the bus to provide the clock signal to allow the slave to transmit data to the master node. Such a mechanism is required mainly because of the structural design of the SPI bus.

The interface between Layer 1 and Layer 2 consists of two ring buffers, one for transmission and one for reception. The two buffers are denominated as *L1_ring_buffer_rx* and *L1_ring_buffer_tx*. The size of the RX and TX ring buffers may be configured separately but in most situations, the size is usually the same for both buffers. The RX ring buffer is written by layer 1 each time a new unstructured raw byte is received over the SPI interface and is read by layer 2. The TX buffer is written by layer 2 and is read by layer 1 each time the latter needs to transmit a raw byte over the SPI interface. For Layer 1, the flow is quite trivial thus there are relatively few operations that need to be done.

In a typical SPI implementation, the software must initially read the data that possibly arrived on the bus and then initiate the transmit procedure of the hardware with the data needed to be transmitted. In some situations, the software module does not have any available new data to be transmitted on the bus. In this case, the software must fill the transmit register of the SPI hardware peripheral with a default byte that is considered to be neutral (in most cases such a byte has the value of 0xFF or 0x00). The PARSECS_RT stack allows the user to set the value of the default neutral byte, but all the nodes connected to the SPI bus must have the same setting. The RX ring buffer is written by Layer 1 when new data arrives on the bus, thus Layer 1 only handles the write pointer of this ring buffer. The TX ring buffer is then checked if any data is available from Layer 2 to be sent to the hardware module to serialize it on the SPI bus. In this case, Layer 1 only handles the read pointer of the TX ring buffer. If any of these buffers are found full when new data has to be inserted then a critical situation occurs which leads to a communication failure. Such a situation may occur only if the stack parameters are not configured properly.

3.2. Layer 2 - Data Link Layer

The second part of the PARSECS_RT Low Level sub-stack is represented by Layer 2, the Data Link Layer. The main role of this layer is to provide a stable, efficient and reliable node-to-node data transfer with flow control, packet acknowledging and error detection features such as a standard data link layer protocol should offer. This layer is divided into two separate modules: one for transmission (Layer 2 TX) and one for reception (Layer 2 RX).

The main interface between Layer 2 and the upper layers is represented by the SPI Base frame in Fig. 3 which is divided into 3 parts: the header, the data field, and the CRC (Cycle Redundancy Check) field. The Header of the SPI Base frame is 4 bytes long and contains the following fields.

- Start Of Frame (SOF): 1 byte long of the strict value of 0x7E which identifies the beginning of an SPI Base Frame
- Length (LEN): 1 byte long which defines the actual length of the data field
- Sequence number (SEQ): 1 byte long which transports the sequence number of an SPI Base Frame
- Type (TYPE): 1 byte long which identifies the type of the SPI Base Frame

As described in Table 1, the SPI Base frame currently defines 3 types of frames, with the possibility for future extensions.

The data field contains the actual payload transported by an SPI Base Frame. The length of this field is defined by the LEN field in the header of the packet, which practically states the value of n – the current number of bytes in the data field. As stated in Fig. 3, the maximum length of this field is given by N_{MAX} which may not exceed the value of 254.

$$n \leq N_{MAX} \leq 254 \tag{1}$$

The CRC field is 2 bytes long and it is used for error detection. The implementation is based on the code and specifications provided by the Internet Request for Comments 1662 (RFC 1662) which describes the PPP in HDLC-like Framing [19]. The CRC value is

Table 1
SPI base frame – TYPE field description.

Frame type	TYPE field value	Description
DATA	0 × 8C	Common data frame — implies having the length field greater than zero
ACK	0 × 7C	Acknowledge frame — implies having the length field equal to zero (LEN = 0), no data is necessary for an acknowledge frame
NACK	0 × 7B	Not-Acknowledge frame — implies having the length field equal to zero (LEN = 0), no data is necessary for a not-acknowledge frame

computed using all the bytes of the SPI Base frame in the header and data field except the SOF byte. The implementation provided by RFC 1662 is based on the following polynomial for CRC16 computation:

$$x^{16} + x^{12} + x^5 + 1 \tag{2}$$

The frame sequencing is implemented using the SEQ field of the header. Each communication partner keeps track of its total number of sent SPI Base frames using a 1 byte counter which is always transported by the SEQ field when transmitting such a frame. On the reception part, the SEQ field is used for packet acknowledging, thus, informing the transmitter that a frame having a certain SEQ number has been acknowledged or not acknowledged. An SPI Base frame is not-acknowledged if the defined rules above are not met: the CRC values calculated by the transmitter and the receiver are different, the maximum value of the LEN field is exceeded, unsupported TYPE field value or the restrictions regarding the LEN field are not met.

The data encapsulation overhead of the SPI Base Frame (E_{L2}) and the value of SPI Base Frame MTU (Maximum Transmission Unit) may be deducted as follows:

$$E_{L2} = SIZE(Header) + SIZE(CRC) = 4 + 2 = 6\,\text{bytes} \tag{3}$$

$$MTU_{L2} = E_{L2} + N_{MAX} = 254 + 6 = 260\,\text{bytes} \tag{4}$$

The data flow between Layer 2 and the upper layers, (in our case, the PARSECS_RT High Level sub-stack) is ensured by a set of buffers that are only accessed by the upper layers through dedicated special APIs. Because Layer 2 has separate flows for transmission and reception, the data flows provide separate buffers. The receiving flow interface is implemented using a set of two reception identical buffers, providing a transparent double buffer solution for the upper layers. This solution ensures no data loss by having a secondary buffer on stand-by until the previously filled buffer is released (read) by the upper layers. Direct access of the upper layers to either the reception buffers or the transmission buffer is prohibited. The only allowed method to access the data is through provided specialized APIs. Such a solution also conceals the buffering mechanisms from the upper layers.

Besides the actual payload, this layer also provides additional information to the upper layers regarding packet sequencing such as

- current sequence number of the data frame (last received data frame sequence number)
- last transmitted data frame sequence number
- last acknowledged data frame sequence number
- last not acknowledged data frame sequence number

The execution and processing flow of this layer is controlled by a series of flags which not only decide this layer's internal operations but also provide the needed multi-threading protection.

The Layer 2 RX component takes the raw unstructured bytes from Layer 1 and assembles the SPI Base Frame and manages the data flow using a double buffering solution. The Layer 2 RX component's execution begins by checking whether there is a reception in progress in either of the two reception buffers (spi_packet_rx_1 and spi_packet_rx_2) by checking the appropriate flags (FG_FRIP_1 and FG_FRIPT_2). In an affirmative case, the assembling process of an SPI Base frame is resumed on the respective buffer. On the other hand, if no reception was found to be in progress on either of the two buffers then an SPI Base Frame assembly is launched on the first free buffer found by checking the responsible flags FG_LL_FUSE_1 and FG_LL_FUSE_2. The SPI Base Frame assembling process may conclude with either the fact that a correct frame has been constructed or that no frame assembly has been finished yet. Such a situation is checked by interrogating flags FG_NFE_1 and FG_NFE_2. If either one of these flags is found to be set, then the corresponding buffer, containing a valid SPI Base Frame, will be processed accordingly.

The frame assembling process is a trivial task, implemented using a simple state machine that mainly checks the correctness of the packets and discards the data in case of an error.

Upon the reception of an SPI Base Frame, layer 2 must signal the proper action to be taken based on the type of frame. In the case of an ACK or NACK frame, this sub-state will only update the sequence numbers of the last acknowledged and not-acknowledged frames. In case a DATA type frame has been received, this sub-state informs the higher levels by setting the FG_HL_NDF_1/2 flag to 1. In all cases, this sub-state concludes its flow by resetting the flag FG_NFE_1/2 informing that the current frame has been processed.

The higher layers, in our case the PARSECS_RT High Level Substack, may interact with the RX flow of Layer 2 through a dedicated API. The Layer 2 RX API begins by checking whether the spi_packet_rx_1 or spi_packet_rx_2 reception buffers have an SPI Base Frame available to be read and released by the higher layers. This situation is identified by API by checking the flags responsible:

FG_HL_NDF_1 and FG_HL_NDF_2. In the case when one of these flags is set, the corresponding buffer containing the SPI Base Frame is read and thus released by the caller. A particular situation is identified when both of these reception buffers contain a valid frame. In such a case the API returns the SPI Base Frame from the buffer that has the lower value of the SEQ field. In any case, the API caller must provide a valid memory zone where the new data will be written. After this operation, the corresponding buffer is released and the lower layers will be notified that it is ready to store a newly received SPI Base Frame.

The transmission component of layer 2 is similar to the reception part. Similar to the Layer 2 RX API, the TX API is also considered only an interface between the Layer 2 TX component and the upper layers and not an actual component of the tasks implementing the PARSECS_RT Low Level Substack. This API is thus called by the higher layers in order to initiate a transmission operation through the PARSECS_RT Low Level Substack.

The Layer 2 TX API begins by checking some of the required flags before trying to initiate a transmission. A transmission is being initiated only if all of the following flags are not set: FG_LL_FTIP, FG_LL_FTT, FG_ACK and FG_NACK. The limitation here is that transmission cannot be initiated if the internal Layer 2 transmission buffer (spi_pack_tx_buffer) is not free, thus, a double buffer solution is not implemented here. In the situation where all the above flags are not set, the transmission buffer is considered to be free and a new transmission request from the upper layers may be accepted. A transmission request involves copying the payload from the memory zone specified by the caller and afterward setting the FG_HL_FTTR flag to the true value. The transmission API caller is then informed about the sequence number (SEQ) that will be assigned to the SPI Base Frame containing its payload. On the other hand, if the transmission buffer is found not to be free and no new transmission requests may be accepted at the moment of the call, the API informs the caller about the busy state.

The main process flow of the Layer 2 TX component has as its main role to serialize the correct SPI Base Frame based on the flag configuration. Before making any new decisions, the first flags that are checked by the main flow of the Layer 2 TX component are FG_LL_FTIP and FG_LL_FTT which signify that a transmission is already in progress or that a frame is already scheduled to be transmitted. In such a case, the process flow executes the actual part which serializes the SPI Base Frame currently in the transmission buffer, into the Layer 1 transmission ring buffer (L1_ring_buffer_tx).

On the other hand, when no transmission is in progress or no other frame is scheduled for transmission, the decision is made based on 3 important flags: FG_ACK, FG_NACK and FG_FTTR. The FG_ACK and FG_NACK flags take precedence. The FG_ACK and FG_NACK flags are used by the RX component of Layer 2 to inform the TX component of Layer 2 that it needs to schedule an acknowledge or not-acknowledge frame transmission. In any of these two situations the corresponding FG_ACK and FG_NACK are reset immediately after a respective frame is scheduled for serialization by setting the FG_LL_FTT flag signifying a frame to transmit is available. If neither of these two flags are found set, the FG_HL_FTTR flag is checked in order to verify if the Layer 2 TX API needs to schedule a transmission of a data type SPI Base Frame. If such a case is identified, the FG_LL_FTT is set in order to signify that a frame to transmit is scheduled along with the reset of FG_HL_FTTR. The main Layer 2 TX component process flow is concluded with the execution of the sub state responsible for the serialization of the scheduled SPI Base Frame. The serialization sub-state of an SPI Base Frame in the Layer 2 TX component is similar to the assembling process flow in the Layer 2 RX component.

The de-serialization process takes the SPI Base Frame from the transmission buffer and disassembles it byte by byte into the Layer 1 transmission ring buffer (L1_ring_buffer_tx). The serialization is only conditioned on the space available in the Layer 1 transmission ring buffer. In the case when this buffer is full, the serialization is suspended and then resumed at the next execution of the task implementing the Layer 2 TX component. When the serialization is finished, the FG_LL_FTIP flag is reset thus signaling that Layer 2 transmission buffer has been released and it is ready to take another SPI Base Frame.

The PARSECS_RT High Level Sub-stack implements the higher layers of the PARSECS_RT SPI communication stack and, in accordance with the OSI Reference Model, provides the following layers: Layer 4 — Transport Layer, Layer 6 — Presentation Layer and Layer 7 — Application Layer. This sub-stack was designed to be implemented as a single atomic task. Regarding the OSI Reference model compatibility and compliance, the PARSECS_RT High Level Sub-stack with the help of the lower levels provides most of the defined functionality.

3.3. Layer 4 — Transport Layer

The Layer 4 — Transport Layer provides a reliable link between any slave node and the master node connected on the same SPI bus in a connectionless-mode network service profile (CLNS) with the features and mechanisms defined by the TP0 protocol class of the OSI Reference Model Layer 4 [17]. This means that this layer provides a connectionless mode communication with PDU segmentation, de-segmentation and reassembly. The transport layer takes its input from the data field of a finite number of SPI Base Frames. Each data field in the SPI Base Frame contains a PARSECS WIT PDU. The transport layer takes such PDUs, assembles them in the correct order, and produces an output represented by a PARSECS WIT FRAME which is interpreted by the presentation layer. The structure of the PARSECS WIT PDU adds a small data encapsulation overhead as it can be observed in Fig. 4

The main role of this layer is to ensure PDU segmentation. A finite number of PARSECS WIT PDU frames are processed by this layer in order to be disassembled and assembled back in the correct order ensuring the formation of a PARSECS WIT FRAME which is then processed by the presentation layer. The structure of the PARSECS WIT FRAME is presented in Fig. 5 where the size of the frame is represented by f and the maximum allowed size if F_{MAX}. with the obvious restriction:

$$f \leq F_{MAX} \tag{5}$$

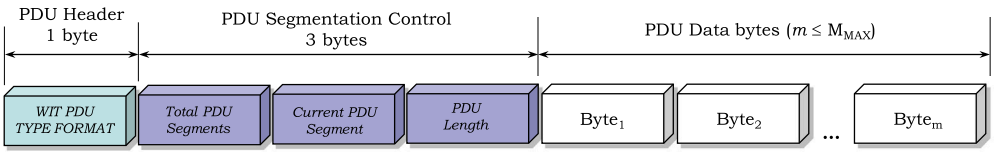


Fig. 4. PARSECS WIT PDU structure.

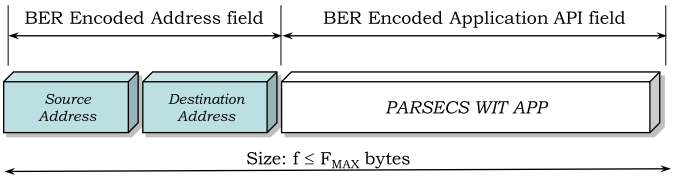


Fig. 5. PARSECS WIT FRAME structure.

The PARSECS WIT PDU has 3 components: a PDU Header field which is 1 byte long, a PDU Segmentation Control field which is 3 bytes long and a PDU Data field. The encapsulation overhead added by the PARSECS WIT PDU is the following:

$$\begin{aligned} E_{L4} &= SIZE(PDU\,Header) + SIZE(SegmentationField) \\ &= 4\,\text{bytes} \end{aligned} \tag{6}$$

Considering (6) along with (3) and Fig. 4 we may state that:

$$M_{MAX} = N_{MAX} - E_{L4} \tag{7}$$

In this case, the maximum value of the PARSECS WIT PDU Data field is limited to 250 bytes as stated in the following statement, where m represents the instant current value of the length of the PARSECS WIT PDU Data field for some such PDU:

$$m \leq M_{MAX} \leq 250 \tag{8}$$

However, there are not other limitations of the maximum value of the PARSECS WIT PDU Data length field beside (8), thus the user may enforce a lower limit in case of hardware constraints.

The PDU segmentation control field is 3 bytes long and contains the information needed to ensure segmentation, de-segmentation and correct data reassembly along, of course, with the SEQ field from the associated SPI Base Frame. The PDU Segmentation control contains the following fields, each 1 byte long.

- TOTAL_SEG – Total PDU Segments – contains the total number of segments in which the data is split
- CUR_SEG – Current PDU Segment – contains the current segment number, from the total number of segments, that is being transferred
- PDU_LEN – PDU Length – contains the length of the PDU Data field of the current PARSECS WIT PDU, in our case a value which is equal to m

The total number of PDU segments, TOTAL_SEG, may be calculated as follows

$$TOTAL\_SEG = \lceil * \rceil \frac{f}{M_{MAX}} \tag{9}$$

In such case, considering (9) naturally there are some restrictions to be defined:

$$TOTAL\_SEG \geq 1 \tag{10}$$

$$CUR\_SEG \geq 1 \tag{11}$$

$$CUR\_SEG \leq TOTAL\_SEG \tag{12}$$

The PARSECS WIT PDU Header is only 1 byte long and contains a bit field denominated as WIT PDU TYPE FORMAT and is meant to be interpreted as a status and error byte. The components of this bit field are presented in Table 2.

The PARSECS WIT PDU is mostly used for data transfer having a length greater than 0 ($m > 0$) and with the WIT PDU TYPE FORMAT field having no error bits set (bits 2, 3 and 7). In such a case, considering that this PDU is transferred by a single SPI Base Frame, the transfer is acknowledged by Layer 2 and no other Layer 4 PDU is transmitted for this purpose. However, in the case when certain errors occur, Layer 4 will respond with a PARSECS WIT PDU transporting the error through the WIT PDU TYPE

Table 2
WIT PDU TYPE FORMAT description.

Bit	Field name	Description
0	MultiPacketPDU	If set to 1 it signifies that the current PDU is part of a segmented transaction
1	LastPDUInMultiPacket	If set to 1 it signifies that the current PDU is the last PDU of a segmented transaction
2	SequenceError	The previous PDU's SEQ value was not correct
3	SizeExceeded	In a segmented transaction, the resulted assembled PDU exceeds the value of F_{MAX}
4–6	Reserved	Bits are reserved for future protocol extension
7	Generic Error	Signifies a generic error different from the error situations already defined above

FORMAT field. Such a PDU is easily recognized by the fact that it only transports the error through the header but no data through the PDU Data field, thus having the PDU LEN (m) equal to 0.

The execution and processing flow of this layer is managed internally by using a data structure containing all the information needed by this layer. The main data structure that this layer of the protocol operates on is described in the following ASN.1 code

```
PARSECS_Protocol_Descriptor ::= SEQUENCE {

ReceivePDU PARSECS_WIT_PDU,
TransmitPDU PARSECS_WIT_PDU,
ReceptionState PARSECS_STATE_DATA,
TransmissionState PARSECS_STATE_DATA,
ErrorTransmitRequired BOOLEAN,
ErrorCode PARSECS_PROTOCOL_STATUS
}
```

Listing 1: PARSECS Protocol Descriptor main data structure

As it can be observed, this layer of the protocol keeps its internal reception and transmission state separate. Such states are described below by a PARSECS_STATE_DATA structure. The protocol needs a *ReceivePDU* and a *TransmitPDU* both representing PARSECS WIT PDU described above Fig. 4, thus this layer operates on separate execution flows for transmission and reception. The structure above also contains an error code if necessary which is transmitted if the Boolean parameter *ErrorTransmitRequired* is set.

```
PARSECS_STATE_DATA ::= SEQUENCE {

ResourceFree BOOLEAN,
IsMultiPacketState BOOLEAN,
IsLastPDUInMultiPacket BOOLEAN,
TotalPDUPackets INTEGER,
CurrentPDUPacket INTEGER,
WITDataLength INTEGER,
CurrentDataIndex INTEGER,
LastSequenceNumber INTEGER,
WITData OCTET_STRING
}
```

Listing 2: PARSECS_STATE_DATA Structure description

The reception and transmission flow state data structures are identical and described above in ASN.1 notation. The meaning of each structure member is the following:

- **WITData** An instance of a PARSECS WIT FRAME in Fig. 5 which represents the interface between the transport layer and the presentation layer. This field is considered an input parameter from the presentation layer for the transmission flow and an output parameter for the reception flow. On the reception flow, this parameter contains the PARSECS WIT FRAME after the correct assembly of multiple PARSECS WIT PDUs of the segmentation/de-segmentation flow. On the transmission flow this parameter contains the PARSECS WIT FRAME that is going to be disassembled by the segmentation and de-segmentation part in order to be transmitted as multiple PARSECS WIT PDUs.
- **WITDataLength** An integer representing the amount of data, in bytes, that is contained in the WITData parameter. The WITDataLength parameter is the same as the one described in (5)
- **IsMultiPacketState** An internal Boolean flag signifying that the RX/TX flow is currently engaged into a segmented reception/-transmission
- **IsLastPDUInMultiPacket** In the situation when the RX/TX flow is in a segmented reception/transmission, this internal Boolean parameter signifies that the current PARSECS WIT PDU is the last PDU in the current segmented reception/transmission
- **TotalPDUPackets** An integer signifying the total number of PDU's involved in the current segmented reception/transmission. In the case when segmentation is not being used this value is always equal to 1
- **CurrentPDUPacket** An integer signifying the current PDU packet number within the segmented transmission/reception. In the case, when the segmentation has not been used, this value is always equal to 1
- **CurrentDataIndex** An integer that represents the index value of the current PARSECS WIT PDU within the PARSECS WIT FRAME

- **LastSequenceNumber** An integer representing the last sequence number of the transmitted/received PARSECS WIT PDU taken from the lower levels of the stack through the SPI Base Frame
- **ResourceFree** A Boolean value specifying that the current transmission/reception flow is free and can take a new operation. This member practically states the availability of WITData

The reception flow of Layer 4 begins by checking whether the reception resource is free (WITData) by checking *ResourceFree*. If the resource is not free the reception process is terminated. On the other hand, if the resource is free, the reception flow calls the API of the lower levels, in this case, the reception API of Layer 2, in order to check if a new SPI Base Frame is available to be processed. If such a frame is available its format is checked prior to being decoded and having the data extracted. If the SPI Base Frame is found to be inconsistent then the whole reception process is aborted. The extracted data from the SPI Base Frame must be in the correct format of a PARSECS WIT PDU. Before continuing with the rest of the flow, the PDU header from the PARSECS WIT PDU is checked for any error flags. In the case when error flags are found to be set then the whole reception process is aborted and all the internal flags are reset. This also implies that a segmented reception that was in progress would be reset.

The main reception flow is, in principle, divided into two sub-flows: a sub-flow when a segmented reception is not in progress and a sub-flow when a segmented reception is currently in progress. This separation of the flow is decided by the internal *IsMultiPacketState* Boolean flag. If the *IsMultiPacketState* flag is false when a new PARSECS WIT PDU is received then no segmented reception is currently in progress. If the newly received PDU also has its *MultiPacketPDU* flag not set (value is false) then no segmented reception will be initiated and the newly received PARSECS WIT PDU represents the only "segment" of data that will be copied at the right position (at CurrentDataIndex 0) within the WITData internal reception buffer. After such a situation, the *resourceFree* internal reception flag is set to true before the termination of the reception flow of this layer so the Presentation Layer, may use the newly received PARSECS WIT FRAME. This whole step is initiated after a check of sequence numbers and parameters as the following:

$$TOTAL\_SEG = CUR\_SEG = 1 \tag{13}$$

$$MultiPacketPDU = FALSE \tag{14}$$

$$LastPDUInMultiPacket = FALSE \tag{15}$$

In the case the above conditions (13), (14) and (15) are not met, the whole reception process is aborted, the received data is discarded, the *IsMultiPacketState* internal flag is being reset to false and an error PDU is scheduled for transmission.

If the *IsMultiPacketState* flag is false and the newly received PDU has its *MultiPacketPDU* flag set (value is true) then a segmented reception will be initiated and the newly received PARSECS WIT PDU represents the first segment of data that will be copied at the right position (at CurrentDataIndex 0) within the WITData internal reception buffer. The internal reception parameters will then be updated as follows: the flag IsMultiPacketState is set to true and the internal reception parameters TotalPDUPackets and CurrentPDUPacket are updated with the values from TOTAL_SEG and respectively CUR_SEG of the header of the newly received PARSECS WIT PDU. Before taking the above actions a consistency check is also being done:

$$TOTAL\_SEG \geq 1 \tag{16}$$

$$LastPDUInMultiPacket = FALSE \tag{17}$$

$$CUR\_SEG + 1 \leq TOTAL\_SEG \tag{18}$$

In the case the above conditions (16), (17) and (18) are not met, the whole reception process is aborted, the received data is discarded, the *IsMultiPacketState* internal flag is being reset to false and an error PDU is scheduled for transmission.

The other part of the flow is represented by the situation when the internal reception flag *IsMultiPacketState* has a true value when a new PARSECS WIT PDU is received. Practically this part of the flow dictates what should be done when a segmented reception is already in progress and a new PDU is received. In such a case, the flow initiates a needed consistency check before performing the required operations:

$$\begin{aligned} &LastPDUInMultiPacket = TRUE \\ &CUR\_SEG = TOTAL\_SEG \end{aligned} \tag{19}$$

$$CurrentDataIndex + m \leq F_{MAX} \tag{20}$$

$$TotalPDUPackets = TOTAL\_SEG \tag{21}$$

$$CurrentPDUPackets = CUR\_SEG \tag{22}$$

Condition (19) states that in the case that the received PARSECS WIT PDU represents the last segment, then the values of the current segment (CUR_SEG) must be equal to the value of the total number of segments (TOTAL_SEG). Condition (20) states that

Table 3
BER general form.

TAG	DATA

the current amount of segmented data already present in the WITData buffer added to the data size of the received PARSECS WIT PDU m does not exceed the size of WITData buffer F_{MAX}. Furthermore, the following conditions (21) and (22) practically verify that the received parameters TOTAL_SEG and CUR_SEG are consistent with the already present values of the reception parameters holding these values (TotalPDUPackets and CurrentPDUPackets). In the case where the above conditions (19), (20), (21) and (22) are not met, the whole reception process is aborted, the received data is discarded, and the IsMultiPacketState internal flag is being reset to false and an error PDU is scheduled for transmission.

After the consistency check has been done successfully, the segmented reception progress is continued. The extracted payload from the newly received PARSECS WIT PDU is being copied to the current position in the WITData buffer represented by the internal reception parameter *CurrentDataIndex*. This value is then updated with the new position where the next segment reception is to be placed.

This part of the flow is concluded by checking the status of the flag *LastPDUInMultiPacket*. If this flag is found to be false then no additional steps are required thus the flow is resumed at the next reception of a PARSECS WIT PDU. On the other hand, if this flag is found to be true then this last received PARSECS WIT PDU represents the last segment. In this situation, it is concluded that a PARSECS WIT FRAME has been assembled in the WITData buffer. The whole reception process is being reset and the Presentation Layer may use the newly assembled PARSECS WIT FRAME.

The transmission flow of the Transport Layer is similar to the reception flow. The transmission flow operates using a data structure identical to the one used by the reception layer (PARSECS_Protocol_Descriptor) keeping the state in an instance of the PARSECS_STATE_DATA structure.

3.4. Layer 6 - Presentation Layer

The next layer of this sub-stack is represented by a Layer 6 – Presentation Layer which, as the OSI Reference model states, serves as an intermediate level between the lower levels and the application. This layer provides data structure representation using mainly Basic Encoding Rules of Abstract Syntax Notation One (ASN.1) Standard [20]. This layer is executed when the user of the stack calls the exported APIs of the stack. The main role of this layer is to provide specific and consistent encoding for the application data. The version of the Basic Encoding Rules that the PARSECS_RT stack uses has the encoding data tags of the DLMS/COSEM (Device Language Message Specification/Companion Specification for Energy Metering) suite which is standardized by the International Electrotechnical Commission [21,22]. The same version of the Basic Encoding rules has also been used in a similar application layer protocol for IoT [23]

The Presentation Layer communicates with the Transport Layer by operating on a specific type of frame designated as PARSECS_WIT_FRAME and having the structure presented in Fig. 5. The PARSECS_WIT_FRAME is a data structure that is encoded using the PARSECS version of the Basic Encoding Rules. Before describing such types of frames, the following paragraphs presents the actual encoding and data type provided by the Presentation Layer, all presented using ASN.1.

According to the standard, the main idea behind the Basic Encoding Rules (BER) may be summarized in the form presented in Table 3

The TAG field represents a byte specifying the type of data that follows in the encoding. The DATA field is dependent on the actual data type. The supported data types, along with the assigned TAG, are presented in ASN.1 notation, under the name of *GenericData*:

```
GenericData ::= CHOICE {
BERNullData              [0]    IMPLICIT NULL,
BERArray                 [1]    IMPLICIT SEQUENCE OF Data,
BERStructure             [2]    IMPLICIT SEQUENCE OF Data,
BERBoolean               [3]    IMPLICIT BOOLEAN,
BERBitString             [4]    IMPLICIT BIT STRING,
BERDoubleLong            [5]    IMPLICIT Integer32,
BERDoubleLongUnsigned    [6]    IMPLICIT Unsigned32,
BEROctetString           [9]    IMPLICIT OCTET STRING,
BERVisibleString         [10]   IMPLICIT VisibleString,
BERInteger               [15]   IMPLICIT Integer8,
BERLong                  [16]   IMPLICIT Integer16,
BERUnsigned              [17]   IMPLICIT Unsigned8,
BERLongUnsigned          [18]   IMPLICIT Unsigned16,
BERlong64                [20]   IMPLICIT Integer64,
BERLong64Unsigned        [21]   IMPLICIT Unsigned64,
BEREnum                  [22]   IMPLICIT Unsigned8,
BERRawData               [254]  IMPLICIT OCTET STRING,
BERDontCare              [255]  IMPLICIT NULLG
}
```

Listing 3: Definition of GenericData

Considering the above definitions, the PARSECS_WIT_FRAME briefly described in Fig. 5 may be defined in ASN.1 with the following content and structure:

```
PARSECS_WIT_FRAME ::= SEQUENCE {
SourceAddress                   BERUnsigned,
DestinationAddress              BERUnsigned,
PARSECS_WIT_APP                 PARSECS_APP_API
}
```

Listing 4: Definition of PARSECS_WIT_FRAME

According to the above definition, the PARSECS_WIT_FRAME is practically implemented using a type of *BERStructure* containing the 3 elements. The *SourceAddress* and the *DestinationAddress* fields are used to provide an addressing mechanism for the user at the application level. Even though for an SPI bus the addressing is implicit through the slave selection lines, such a mechanism is intended to be used at the application level. As a convention, the PARSECS_RT stack considers the master node to have the address 0. In this case, the *SourceAddress* and *DestinationAddress* fields designate the source and the destination of the current PARSECS_WIT_FRAME. The PARSECS_WIT_APP field is considered to be of type PARSECS_APP_API which is practically implemented as a *BERStructure* data type containing application layer specific elements.

3.5. Layer 7 — Application Layer

The final layer of the stack is represented by the Application Layer which provides a unified homogeneous API, based on the request–response paradigm, which may be used to transfer application specific data between communicating nodes and to also call application specific remote methods similar to a remote procedure call mechanism. The Application Layer is designed to be used on top of the Presentation Layer and not separate from the PARSECS_RT High Level Sub-stack.

The Application Layer is based on a request–response paradigm. Each node connected to the SPI bus may request or respond (to) information to/from any other node regardless of its place as master or slave on the bus. Practically, on the application layer the nodes are considered equals. The request–response paradigm of the PARSECS_RT Application Layer operates either on parameters that may be read or written or on remote methods which, of course, may only be called. The latter implements a basic remote procedure call mechanism. The parameters are used to transmit information and the methods are used to call a specific action of a node. In this case, the parameters may be read (get) and written (set) while the methods are called.

All of the operations that are offered by the Application Layer are transported by the PARSECS_WIT_APP field implemented as a PARSECS_APP_API as described below:

```
PARSECS_APP_API ::= SEQUENCE {
OperationType           OPERATION_TYPE,
OperationControl        OPERATION_CONTROL,
TypeID                  BERUnsigned,
OperationData           GenericData
}
```

Listing 5: Definition of PARSECS_APP_API

The first component, *OperationType*, defines the type of the transaction and it is described below:

```
OPERATION_TYPE ::= ENUMERATED {
GetRequest                      (1),
GetResponse                     (2),
SetRequest                      (3),
SetResponse                     (4),
CallRequest                     (5),
CallResponse                    (6),
}
```

Listing 6: Definition of OPERATION_TYPE

As it can be observed above, practically, the PARSECS Application Layer supports 3 transaction types defined by enumerated values (1), (3) and (5). The supported transactions offer support for requesting the value of a parameter (1), setting the value of a parameter (3) and calling a remote action method (5). Each of these possible 3 transactions also implies a response for each request: a response when a parameter value is requested (2), a response when a parameter value change is requested (4) and a response when a remote method is being called (6). The parameter of which value is being requested (1) or which value change is being requested (3) or the method which is being called (5) are all identified by an 8 bit unsigned integer implemented as a *BERUnsigned* and denominated as *TypeID* in the structure of PARSECS_APP_API. This field provides a unique identification of a parameter or method. The *OperationControl* is dependent on the type of message (as of being a request or response type message) and is described below as an *OPERATION_CONTROL* construction:

```
OPERATION_CONTROL ::= CHOICE {
NoData                  [0]     BERNullData,
OperationResult [1]     OPERATION_RESULT,
}
```

Listing 7: Definition of OPERATION_CONTROL

In the case when the message transports a request operation type (a *GetRequest* – 1, a *SetRequest* – 3 or a *CallRequest* – *5*), the *OperationControl* field is implemented as a *BERNullData*. In this case, this field is practically not needed, but to maintain homogeneity, this field is implemented as a data type that transports no data. On the other hand, in the case when the message transports a response operation type (a *GetResponse* – 2, a *SetResponse* – 4, or a *CallResponse* – 6), the *OperationControl* field is implemented as a type of *OPERATION_RESULT* containing the result of the requested operation

```
OPERATION_RESULT ::= ENUMERATED {
Success                         (0),
HardwareFault                   (1),
TemporaryFailure                (2),
ReadDenied                      (3),
WriteDenied                     (4),
ParameterMethodUndefined        (5),
OperationTimeout                (6),
ParameterSyntaxError            (7),
OperationUnsupported            (8),
AddressMismatch                 (9),
OtherReason                     (254),
UnknownValue                    (255),
}
```

Listing 8: Definition of OPERATION_RESULT

In the above Listing 8 the operation result is represented as an enumeration of some predefined operation result reasons, but also leaving space for user and future extensions (IDs 10-253). This field is implemented using a data type of *BEREnum*.

The data field of the message described in 5 that transports the actual data of the application layer is represented by *OperationData*. The type of this field is represented by a *GenericData* type. The data that is transported by the *OperationData* field is dependent on the application and it is used in concordance with the type of the PARSECS_APP_API message.

In case the transported application layer message is of type Get Request, the sender intends to request a parameter value from the destination node. In such case, the encoded message is implemented as a *BERStructure* according to Listing 5. The OperationType filed is encoded as a *BEREnum* type of value 1. The *OperationControl* field technically has no meaning in this case and is only present here for reasons like predictability, compatibility, and homogeneity. The value of this field is implemented as a *BERNullData* type which practically consists of a single byte of value 0. The *TypeID* filed will contain the ID of the requested parameter and is encoded as a *BERUnsigned* type. The last field, *OperationData*, has no meaning in this context, it is kept here again for reasons like predictability, compatibility and homogeneity similar to the *OperatingControl*. For this reason, the value of this field is implemented as a *BERNullData* type. In very few words, a Get Request message is intended for getting the value of a parameter identified by the field *TypeID*.

In the above context, the response given by the node receiving a Get Request message is implemented by a message of type Get Response. The implementation is similar to the previous one, beginning with an *OperationType* filed encoded as a *BEREnum* type of value 2. In this case, the *OperationControl* has a dedicated meaning and transports the result of the request message, whether it was a success or an error. In the latter case, the reason for the error is transported in this field according to Listing 8. The *TypeID* field is also transported here with the same value as the value from the request message providing a simple verification mechanism of data correctness. The *OperationData* field contains the actual value of the requested parameter encoded as one of the available data types provided by *GenericData* in Listing 3. The data in this field is available only if the operation was a success, thus the *OperationControl* field has the value of Success implemented as a *BEREnum* of value 0. On the other hand, if the *OperationControl* transports an error then the *OperationData* type transports a *BERNullData*.

The Set Request message type is used by one node to request the change of a parameter of another node. Most of the structure and meaning of the fields are similar to the ones described for the Get Request message type with the obvious difference that the *OperationType* is implemented as a *BEREnum* of value 2. Another notable difference here is that the OperationData is not implemented as a *BERNullData* thus this field must contain the new value of the parameter that needs to be changed. In this context, the value of *OperationData* has a form described by the type *GenericData*.

The obvious response of a Set Request message is a Set Response message practically notifying the requesting node of the result of the parameter change operation. The structure and meaning of such a message are practically identical to the ones described for the Get Response message.

The Call Request message type may be used by a node on the network to request the execution of a remote method or action of some other node connected to the same network. The structure and meaning of the fields are similar to the ones described for the *GetRequest* message with the obvious difference that the OperationType is implemented as a *BEREnum* of value 4. Another small but notable difference is the fact that the *TypeID* fields here represent the identification number of the remote method that is requested to be called. Furthermore, in contrast with the Get Request message, for the Call Request message, the *OperationData* field is not necessarily needed to be implemented as a *BERNullData*. The reason is that for a Call Request message, this field transports the remote method's arguments implemented as one of the data types provided by *GenericData*. This field may be implemented as a *BERNullData* only if the method does not require any arguments.

The natural response to a Call Request message is the Call Response message. The structure and meaning of the fields are almost identical to the ones described for the Get Response message with the obvious difference that the *OperationType* is implemented as a *BEREnum* of value 6. Some other small but notable differences are that the *TypeID* transports the ID of the method that has been

called and its call result is transported by the current Call Request message and also that the *OperationData* transports the return value of the remote method implemented as one of the provided types by *GenerticData*.

The Call Request and Call Response messages practically implemented a very simple, efficient, and general remote procedure call mechanism.

All of the aspects presented in this section are limited to the situation where the communication is made by a master node with a single slave node, but only to facilitate the description of the protocol and to provide concise explanations. However, the PARSECS_RT stack was designed to support a large number of slave modules connected to the SPI bus. The only limitations here are in terms of memory and execution performance of the master node along with the availability of slave selection lines.

4. Real-time analysis and time constraints

The PARSECS_RT Stack was designed not only to provide full stack communication on the SPI bus but to also provide predictable hard real-time communication. The underlying level of the stack, the SPI bus is known for its hard real-time and predictable behavior mainly because of its synchronous and time-triggered architecture. In order to provide the hard real-time and predictability features it was designed for, the PARSECS_RT stack implementation requires the existence of a real-time operating system such as FreeRTOS, Haretick [24], Litmus RT... etc. This section aims at presenting a formal real-time analysis of the PARSECS_RT stack.

Before defining the formal time analysis, we have established some considerations:

- The implementation of the PARSECS_RT Low Level sub-stack is designed as a single atomic task containing the execution of all the layers of the sub-stack (L1 – TX/RX, L2 – TX, L2 – RX). The task is denominated as PARSECS Low Level Task
- One execution of the PARSECS Low Level Task is able to receive and transmit 1 byte (no hardware improvements such as FIFO or DMA are considered)
- The implementation of the PARSECS_RT High Level sub-stack is designed as a single atomic task containing the execution of all the layers of the sub-stack (L4, L6). The task is denominated as PARSECS High Level Task
- The Application Layer (L7) is considered as a call or task of the user's application and is not a complete part of this analysis.
- The SPI clock is considered much higher than the actual task execution frequency, thus an instant transmission on the SPI line compared to the frequency of the task execution.

Furthermore, the following terms are defined:

- $T_{P\_LL\_S}$ - execution period of the PARSECS Low Level Task for a slave node
- $F_{P\_LL\_S}$ - execution frequency of the PARSECS Low Level Task for a slave node

$$F_{P\_LL\_S} = \frac{1}{T_{P\_LL\_S}} \tag{23}$$

- $T_{P\_LL\_M}$ - execution period of the PARSECS Low Level Task for a master node
- $F_{P\_LL\_M}$ - execution frequency of the PARSECS Low Level Task for a master node

$$F_{P\_LL\_M} = \frac{1}{T_{P\_LL\_M}} \tag{24}$$

- $T_{P\_HL\_S}$ - execution period of the PARSECS High Level Task for a slave node
- $T_{P\_HL\_M}$ - execution period of the PARSECS High Level Task for a master node

As it has been stated before, at each execution of the Layer 1 component, the master node polls each connected slave node for transmission and reception thus providing the necessary clock signal on the SPI clock line for each connected slave. Such a statement implies that the whole timing of the communication is practically dictated by the execution period of the PARSECS Low Level Task of the master node, respectively the $T_{P\_LL\_M}$ time parameter.

The first time parameter that is dictated by $T_{P\_LL\_M}$ is the execution period of the PARSECS Low Level Task for a slave node, $T_{P\_LL\_S}$. In order to ensure no data loss the Nyquist theorem has to be applied thus obtaining the fact that the execution frequency of the PARSECS Low Level Task for a slave node must be at least twice the execution frequency of the PARSECS Low Level Task defined by the master node, thus obtaining:

$$F_{P\_LL\_S} \geq 2 \cdot F_{P\_LL\_M} \tag{25}$$

Applying (25) using (24) and (23), the execution period of the PARSECS Low Level Task of the slave node need to respect the following condition:

$$T_{P\_LL\_S} \leq \frac{T_{P\_LL\_M}}{2} \tag{26}$$

The easiest and most practical way would be to consider the lower limit for the $T_{P\_LL\_S}$ parameter:

$$T_{P\_LL\_S} = \frac{T_{P\_LL\_M}}{2} \tag{27}$$

The above restrictions must be properly enforced by the underlying real-time operating system in order for the PARSECS_RT Low Level Substack to provide fully predictable and real-time communication.

The timings of the PARSECS_RT High Level Substack are completely dependent on the time parameters of the PARSECS_RT Low Level Substack. According to the description of the SPI Base Frame in Fig. 3 and as calculated in (3) we can state that the minimum size of such a frame is 6 bytes. Considering this information, in order to ensure that there is no loss of information, the task that implements the PARSECS_RT High Level Substack must have an execution period that is not longer than at most 6 executions of the PARSECS_RT Low Level Substack. Practically an execution period of the PARSECS_RT High Level task must not contain more than 6 executions of a PARSECS_RT Low Level task. Such an assessment is applied to both master and slave implementations, thus, anyway, these are considered identical. These statements can be summarized as follows:

$$T_{P\_HL\_M} \leq 6 \cdot T_{P\_LL\_M} \tag{28}$$

$$T_{P\_HL\_S} \leq 6 \cdot T_{P\_LL\_S} \tag{29}$$

Even though in real-time communication, bandwidth is not of utmost importance in contrast with predictability, still such a parameter is mandatory to be expressed. The raw transfer bandwidth is a trivial matter and depends on the performance of the system that executes the implementation. In this case, the raw bandwidth can be deducted from the timings of the PARSECS_RT Low Level Substack and may be expressed in bps as follows:

$$B_{raw}\ [\text{bps}] = 8 \cdot \frac{1}{T_{P\_LL\_M}} \tag{30}$$

The actual transfer rate for the PARSECS_RT Low Level Substack must also take into consideration the encapsulation overhead as defined in (3)

$$B_{LL}\ [\text{bps}] = 8 \cdot \frac{1}{T_{P\_LL\_M}} \cdot \frac{1}{E_{L2}} \tag{31}$$

Considering that we have a hard real-time communication, the latency shall be perfectly predictable and calculable. In the situation where there is no packet segmentation, we may state that a maximum, one way single transfer latency is defined as follows:

$$L_{MAX} = MTU_{L2} \cdot T_{P\_LL\_M} + T_{P\_HL\_M} \tag{32}$$

Of course, considering a full transaction, representing a request and a response, the latency of the whole transaction would naturally be:

$$L_{MAX} = 2 \cdot (MTU_{L2} \cdot T_{P\_LL\_M} + T_{P\_HL\_M}) \tag{33}$$

In the situation where packet segmentation intervenes, the calculations in (32) and (33) would be multiplied by the number of segments needed.s

In terms of packet loss, the PARSECS_RT stack does not handle this situation in this current version, mainly because data loss does not occur on the SPI bus.

All of the above calculation and explanations were made considering that the communication takes part between one master node and only one slave node. This consideration was taken only to facilitate the description in order to provide a clear view. Naturally, as state before in this paper, the PARSECS_RT stack was designed to handle more than one slave device. The addition of a slave device has minimum impact on the implementation of the stack especially regarding the real-time aspects. The only impact appears at the master side and influences the Worst Case Execution Time (WCET) of the tasks implementing the stack. This can lead to a more complex task scheduling operating and it may even result in a failure to schedule the tasks in the situation when the master node is extremely low on hardware resources. Also, it is important to mention that the addition of slave devices has absolutely no impact on the implementation of the stack on the slave side.

5. Experimental results

We implemented PARSECS_RT on two different platforms both being part of a WIT (Wireless Intelligent Terminal), a node of the CORE_TX project (Collaborative Robotic Environment - The Timisoara eXperiment) [3]. We will continue having the same considerations as described in the previous section, thus having PARSECS_RT Low Level sub-stack and PARSECS_RT High Level sub-stack respectively implemented as two independent and atomic tasks. The periods of the tasks are presented in Table 4.

The slave part of PARSECS_RT bus is implemented on the communication module of the WIT having its CPU based on ARM7TDMI-S architecture [25], i.e. a microcontroller from the NXP LPC2000 family [26]. The operating system running on the communication module is represented by our hybrid implementation [27] of the HARETICK (Hard Real-Time Compact Kernel) [28] kernel along with FreeRTOS [29], thus taking advantage of the jitterless scheduling provided by FENP (Fixed Execution Non-Preemptive) [30] along with the flexibility provided by FreeRTOS.

On the communication board, there are 10 running tasks: 8 tasks running in the FreeRTOS context and two tasks being scheduled by the Haretick context using the FENP algorithm. The two tasks running in the FENP context are the PARSECS Low Level Task and another task designated as XBEE. Considering (1) from [30]:

$$\mu_i = \{T_i, C_i\} \tag{34}$$

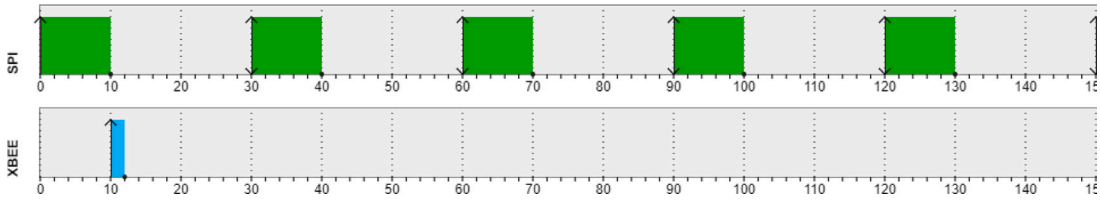


Fig. 6. Task 1 mapping on task 2 execution period.

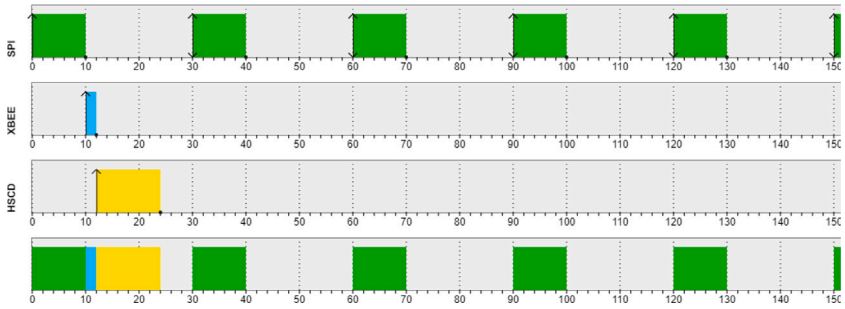


Fig. 7. Task execution mapping using FENP.

Table 4
Test case parameters.

Parameter	Value	Parameter	Value
$T_{P\_LL\_S}$	300 µs	$T_{P\_LL\_M}$	600 µs
$T_{P\_HL\_S}$	1 ms	$T_{P\_HL\_M}$	1 ms
Slave CPU Freq	58.9824 MHz	SPI SCLK	1.56 MHz
B_{raw} [bps]	13.33 kbps	B_{LL} [bps]	2.22 kbps

We define the FENP set of tasks according to the test parameters in Table 4 as

$$\begin{cases} \mu_1 = \{300\ \mu s, 100\ \mu s\} \\ \mu_2 = \{1500\ \mu s, 20\ \mu s\} \end{cases} \tag{35}$$

where μ_1 represents the task parameters for the PARSECS Low Level Task and μ_2 is the parameter for the other FENP task in the system (XBEE). The task parameters are T_i is the task execution period, which in our case is equal to the task deadline, and C_i represents the WCET (Worst Case Execution Time) of the task. In order to apply the FENP scheduling algorithm, the set of tasks needs to be in accordance with the rules defined in (3) from [30]:

$$\begin{cases} S^{FENP} \equiv \{\mu_1, \mu_2, \mu_3\} \\ \mu_i = (\phi_i, T_i, C_i) \\ T_i \leq T_{i+1} \\ S_{i,k+1} = S_{i,k} + T_i = \phi_i + kT_i, \forall \mu_i \in, S^{FENP}, k \in \mathbb{N} \end{cases} \tag{36}$$

where μ_1 represents the PARSECS Low Level Task, μ_2 the XBEE task, μ_3 the HSCD (scheduler task) and ϕ_i the relative phase or the start time as defined in (3) from [28] which will be determined as follows:

1. determine the common divider $GCD(T_1, T_2) = 300$
2. determine the least common multiplier $LCM(T_1, T_2) = 1500$
3. map the execution of the first task (μ_1) over the period of the second task (μ_2) - μ_1 is executing five times within the period of μ_2 as shown in Fig. 6, generated using SimSo Simulator [31], where the scale is 1:10.
4. and determine the first free interval for a complete execution of μ_2 which in this case is [100; 120]
5. determine the execution parameters of the HSCD (task scheduler), considering the HSCD is task μ_3 and $C_3 = 120$ µs

As shown in Fig. 6, T_1 is a multiple of T_2, thus the scheduling cycle is equal to T_1. For (μ_3) we repeat steps 1–5 against each of the other two tasks (μ_1, μ_2) resulting in an execution mapping as in Fig. 7, generated using SimSo Simulator [31]. It is important to mention that the simulator was used here only to validate the actual schedule that was then applied to the hardware.

The master component of PARSECS_RT is implemented on the main module of the WIT referred to as the WIT's motherboard. From a hardware point of view, the motherboard is implemented mainly using the popular Raspberry PI3 [32]. The operating

Fig. 8. PARSECS task executions on hardware.

```
SPI <- MASTER seq: 219 size:  39<LF>
SPI :: seq  TX - 0, RX - 219, ACK - 0, NACK - 0<LF>
SPI_WIT <- F F F F Total:   1 Current:   1 PDUSize:  35<LF>
SPI USER <- 00 - 06 Call Request -Success- Type: 15 Data: <LF>
SPI USER -> 06 - 00 Call Reponse -Temporary failure- Type: 15 Data: <LF>
SPI :: seq  TX - 0, RX - 0, ACK - 0, NACK - 0<LF>
SPI_WIT -> F F F F Total:   1 Current:   1 PDUSize:  15<LF>
SPI -> MASTER seq:   1 size:  19<LF>
```

Fig. 9. PARSECS communication log capture.

system is provided by ArchLinux [33] for ARM having added the Litmus^{RT} kernel patch [34] in order to provide HRT support. The PARSECS application is executed along the other Linux System processes and it is divided into 6 different threads. Each thread is scheduled using Litmus^{RT}, having its dedicated reservation. The threads which are responsible for implementing the PARSECS_RT communication tasks are scheduled with the parameters described in Table 4.

A capture of the executing tasks implementing the PARSECS_RT communication stack may be found in Fig. 8, capture performed using Saleae Logic Analyzer, Logic 2.4.6 [35]. This figure illustrates the jitterless execution of the FENP tasks. A predictable and jitterless communication is one of the key aspects in real-time systems in general and RT-IoT in particular.

In Fig. 9 we present a full stack message exchange between the master and the slave. This capture represents a capture of the logging system of the slave module taken using Docklight Scripting [36] during intense communication. The first line of the log represents the arrival of an SPI Base Frame from the *MASTER* with a sequence number of 219 and a size of 39 bytes. The second line represents a status line with displays the current transmission and reception sequence numbers and the number of frames in the queue that need to be acknowledged (or not). The third line represents a PARSECS WIT PDU (written as *SPI_WIT*) that has been decoded from the SPI Base Frame. Here we may easily identify the 4 bytes of encapsulation of the PARSECS WIT PDU, designated as E_{L4} in (6). Finally, in the 4th line, we may identify the further decoded data as a PARSECS WIT Frame designed here as *SPI USER*. From this line, we may identify that, at the application level, a Call Request was emitted. The following 4 lines represent the response transmitted by the slave module, in this case a Call Response with a Temporary Failure response. The following lines are similar to the first lines with the observation that they are in reverse order, thus being transmitted.

After providing the mathematical proof that the real-time requirements are met, originated from model (36), we validated our work in a real life environment. We employed the same platform described at the beginning of this section (i.e., an ARM7TDMI-S based architecture for the slave device and a Raspberry PI running the Litmus^{RT} extension for the Linux kernel) for extensive experiments during a time interval of three weeks, which provided the expected results.

6. Discussion and future directions

6.1. Comparative analysis

PARSECS_RT is designed to be a modular full stack protocol especially for low-end devices (but not necessarily limited to) for the SPI bus which provides a support for hard real-time communications. The closest available similar solution is the 1-Wire bus which was mainly designed to support and incorporate specific devices from Maxim Dallas such as temperature sensors, iButton devices, small memories, power supplies. Even though it offers a full stack hard-real time communication, it is restricted to very limited functionalities thus being intended to support only specific dedicated devices. Another worth mentioning solution is the CAN bus, designed mainly for automotive communication, which offers only limited real-time support [37,38] (mainly only for a strict prioritized communication). Our solution aims at solving some of these drawbacks and providing a flexible solution initially for SPI communication but with a high degree of adaptability for other communication interfaces and buses. A brief comparison may be found in Table 5 that supports this assessment.

Table 5
PARSECS_RT comparison.

	Stack layers	Real-time support	Device types	Modular
1-Wire	1,2,3,4 (limited)	HRT	Only specific devices	No
CAN	1,2,4,7	limited real-time	Any - with CAN bus interface	No
I2C	1,2	HRT	Any - with I2C bus interface	No
PARSECS_RT	1,2,4,6,7	HRT	Any - with SPI bus interface	No

6.2. PARSECS_RT generalizability aspects

The PARSECS_RT protocol stack is divided into two independent sub-stacks. The PARSECS_RT Low level Sub-stack covers the first two layers of the OSI Model while the PARSECS_RT High Level Sub-stack handles the other upper layers of the stack. Its main design destination is intended for low-end devices. However, giving its modular design, it can be adapted to accomplish real-time communication over other buses than SPI such as I2C [39], LIN [40] or even RS-485 [41]. Our solution's applicability may be considered for a much wider set of applications by using different underlying buses.

The design of PARSECS_RT is not specifically aimed for a specific hardware platform or operating system, thus we have considered a platform independent approach. As we stated in the previous sections of this paper, we used our solution on two different hardware platforms with various operating systems thus proving its effectiveness for a general communication solution where such an option is not currently available.

6.3. Limitations

As we discussed in the previous section, our solution is currently only available for SPI as an underlying bus but the design permits a customization for different other physical communication mediums. In this regard, an important limitation of the stack is that it does not directly address the data loss concerns. The main reason for this derives from particular stack architecture design on top of the SPI bus, which inherently considers that message loss cannot be raised if the bus is correctly used.

Another missing piece of our solution is a network layer to provide routing capabilities. Such an aspect could provide a much more flexible communication by enabling slave devices to communicate between each other.

While considering extensions of PARSECS_RT to other underlying dynamic buses, the necessity for adding the currently missing session layer would naturally arise. Currently such a layer is not available only due to the rigid construction of the SPI bus where the nodes are always connected thus the sessions may be considered infinite.

6.4. Future work

We intend to adapt and evaluate the PARSECS_RT stack using different underlying buses such as the previously mentioned (i.e., CAN, I2C, LIN, etc.). We also plan to improve the proposed protocol by adding retransmission mechanisms in order to recover from message loss. This should not be an overwhelming task considering the fact that the stack already processes sequence numbers for the data transfer.

As another future extension, we plan to add a network layer to the stack in order to provide data routing between slaves. In this approach, we intend to enable the possibility of one slave device to send data directly to another slave device, such an operation generally requiring the intervention of the master node to route the information.

Furthermore, we intend to provide a way to negotiate the message/packet size limits. Currently, communication parameters such as F_{MAX}, M_{MAX} or N_{MAX} need to be the same on all nodes thus implying that the whole communication needs to be adapted to the weakest node. In our perspective, this can be done at the master node level by negotiating the communication parameters with the connected slave nodes. This way, the master may use different values for each slave node and not global values as it uses now.

7. Conclusions

In this paper, we propose PARSECS_RT, a real-time predictable and jitterless communication protocol for wired intra-node communication, particularly suited for time-constrained IoT networks. The correspondence between the OSI reference levels and our proposed implementation of PARSECS_RT is thoroughly described and a real-time and task set feasibility analysis is also provided. Protocol analysis and extensive experimental tests in real life scenarios proved its effectiveness and viability.

CRediT authorship contribution statement

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

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] M. Szalay, P. Matray, L. Toka, Real-time faas: Towards a latency bounded serverless cloud, IEEE Trans. Cloud Comput. (2022).
[2] N. Promwongsa, A. Ebrahimzadeh, D. Naboulsi, S. Kianpisheh, F. Belqasmi, R. Glitho, N. Crespi, O. Alfandi, A comprehensive survey of the tactile internet: State-of-the-art and research directions, IEEE Commun. Surv. Tutor. 23 (1) (2020) 472–523.
[3] R.-D. Cioarga, M.V. Micea, B. Ciubotaru, D. Chiuciudean, D. Stanescu, CORE-TX: Collective robotic environment-The timisoara experiment, in: Proceedings of the Third Romanian-Hungarian Joint Symposium on Applied Computational Intelligence, SACI, 2006.
[4] P. Gaur, M.P. Tahiliani, Operating systems for IoT devices: A critical survey, in: 2015 IEEE Region 10 Symposium, IEEE, 2015, pp. 33–36.
[5] M.V. Micea, HARETICK: A real-time compact kernel for critical applications on embedded platforms, in: Proc. 7th Intl. Conf. Development and Applic. Syst., DAS, 2004, pp. 16–23.
[6] B.-S. Kim, H. Park, K.H. Kim, D. Godfrey, K.-I. Kim, A survey on real-time communications in wireless sensor networks, Wirel. Commun. Mobile Comput. 2017 (2017).
[7] G. Franchino, G. Buttazzo, A power-aware MAC layer protocol for real-time communication in wireless embedded systems, J. Netw. Comput. Appl. 82 (2017) 21–34, http://dx.doi.org/10.1016/j.jnca.2017.01.006.
[8] P. Bartolomeu, M. Alam, J. Ferreira, J. Fonseca, Survey on low power real-time wireless MAC protocols, J. Netw. Comput. Appl. 75 (2016) 293–316, http://dx.doi.org/10.1016/j.jnca.2016.09.004.
[9] C. Bayılmış, M.A. Ebleme, Ü. Çavuşoğlu, K. Küçük, A. Sevin, A survey on communication protocols and performance evaluations for Internet of Things, Digit. Commun. Netw. (2022).
[10] L. Beltramelli, A. Mahmood, P. Österberg, M. Gidlund, P. Ferrari, E. Sisinni, Energy efficiency of slotted LoRaWAN communication with out-of-band synchronization, IEEE Trans. Instrum. Meas. 70 (2021) 1–11, http://dx.doi.org/10.1109/TIM.2021.3051238.
[11] M.K. Pedhadiya, R.K. Jha, H.G. Bhatt, Device to device communication: A survey, J. Netw. Comput. Appl. 129 (2019) 71–89, http://dx.doi.org/10.1016/j.jnca.2018.10.012.
[12] D. Awtrey, D. Semiconductor, Transmitting data and power over a one-wire bus, Sensors- J. Appl. Sens. Technol. 14 (2) (1997) 48–51.
[13] C.-W. Lin, A. Sangiovanni-Vincentelli, Cyber-security for the controller area network (CAN) communication protocol, in: 2012 International Conference on Cyber Security, IEEE, 2012, pp. 1–7.
[14] T. Pop, P. Pop, P. Eles, Z. Peng, A. Andrei, Timing analysis of the FlexRay communication protocol, Real-time Syst. 39 (2008) 205–235.
[15] M.V. Micea, G.N. Carstoiu, L. Ungurean, D. Chiciudean, V. Cretu, V. Groza, Predictable data communication interface for hard real-time systems, in: 2008 International Workshop on Robotic and Sensors Environments, 2008, pp. 98–101, http://dx.doi.org/10.1109/ROSE.2008.4669188.
[16] M.V. Micea, G.N. Carstoiu, L. Ungurean, D. Chiciudean, V.-I. Cretu, V. Groza, PARSECS: A predictable data communication system for smart sensors and hard real-time applications, IEEE Trans. Instrum. Meas. 59 (11) (2010) 2968–2981, http://dx.doi.org/10.1109/TIM.2010.2046363.
[17] H. Zimmermann, OSI reference model - The ISO model of architecture for open systems interconnection, IEEE Trans. Commun. 28 (4) (1980) 425–432, http://dx.doi.org/10.1109/TCOM.1980.1094702.
[18] A.H.M. Aman, E. Yadegaridehkordi, Z.S. Attarbashi, R. Hassan, Y.-J. Park, A survey on trend and classification of internet of things reviews, Ieee Access 8 (2020) 111763–111782.
[19] W. Simpson, PPP in HDLC-like Framing, Tech. Rep., 1994.
[20] 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.
[21] International Electrotechnical Commission, Electricity metering data exchange - The DLMS/COSEM suite - Part 1-0: Smart metering standardisation framework, 2014.
[22] International Electrotechnical Commission, Electricity metering data exchange - The DLMS/COSEM suite - Part 5-3: DLMS/COSEM application layer, 2016.
[23] V. Stangaciu, M. Stanciu, L. Lupu, M.V. Micea, V. Cretu, Application layer protocol for IoT using wireless sensor networks communication protocols, in: 2017 9th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops, ICUMT, 2017, pp. 430–435, http://dx.doi.org/10.1109/ICUMT.2017.8255160.
[24] C.S. Stangaciu, M.V. Micea, V.I. Cretu, Hard real-time execution environment extension for FreeRTOS, in: 2014 IEEE International Symposium on Robotic and Sensors Environments (ROSE) Proceedings, 2014, pp. 124–129, http://dx.doi.org/10.1109/ROSE.2014.6953035.
[25] T. Martin, The Insider's Guide to the Philips ARM7-Based Microcontrollers, Hitex, UK, Ltd, Coventry, 2005.
[26] NXP Semiconductors, User Manual, Tech. Rep., Koninklijke Philips Electronics N.V, 2004.
[27] C. Stangaciu, M. Micea, V. Cretu, An analysis of a hard real-time execution environment extension for FreeRTOS, Adv. Electr. Comput. Eng. 15 (3) (2015) 79–87.
[28] M.V. Micea, V. Cretu, V. Groza, Predictable signal generation with the hard real-time operating kernel HARETICK, in: 2005 IEEE Instrumentationand Measurement Technology Conference Proceedings, Vol. 3, IEEE, 2005, pp. 2097–2102.
[29] R. Barry, FreeRTOS Reference Manual: API Functions and Configuration Options, Real Time Engineers Limited, 2009.
[30] M.V. Micea, C.S. Stangaciu, V. Stangaciu, V.I. Cretu, Improving the efficiency of highly predictable wireless sensor platforms with hybrid scheduling, in: 2012 IEEE International Symposium on Robotic and Sensors Environments Proceedings, IEEE, 2012, pp. 73–78.
[31] SimSo - Simulation of multiprocessor scheduling with overheads, 2023, https://projects.laas.fr/simso/, Accessed: 2023-03-03.
[32] Raspberry PI, Datasheet raspberry PI 3 model B technical specification, [Online], http://www.farnell.com/datasheets/2027912.pdf.
[33] J. Dieguez Castro, Arch linux, in: Introducing Linux Distros, A Press, Berkeley, CA, 2016, pp. 235–252.
[34] 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.
[35] Saleae logic analyzers, 2023, https://www.saleae.com, Accessed: 2023-03-03.
[36] Docklight scripting, 2023, https://docklight.de/, Accessed: 2023-03-03.
[37] L.M. Pinho, F. Vasques, Reliable real-time communication in CAN networks, IEEE Trans. Comput. 52 (12) (2003) 1594–1607.

[38] Tindell, Hansson, Wellings, Analysing real-time communications: controller area network (CAN), in: 1994 Proceedings Real-Time Systems Symposium, IEEE, 1994, pp. 259–263.
[39] J. Valdez, J. Becker, Understanding the I2C Bus, Texas instruments, 2015.
[40] M. Ruff, Evolution of local interconnect network (LIN) solutions, in: 2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484), Vol. 5, IEEE, 2003, pp. 3382–3389.
[41] Z. Shang, Z. L.I., Q. Wei, S. Hao, Livestock and poultry posture monitoring based on cloud platform and distributed collection system, Internet Things (2024) 101039, http://dx.doi.org/10.1016/j.iot.2023.101039, URL https://www.sciencedirect.com/science/article/pii/S2542660523003621.