University of Padova

Department of Information Engineering

Master Degree in Computer Engineering

Porting of the 802.15.4 stack on FreeRTOS in Asymmetric Multi Processing contexts for IoT solutions

Federico Munari

Supervisor:

Prof. Michele Moro

Academic year 2017-2018

Sommario

L’obiettivo di questa tesi è effettuare il porting dello stack IEEE 802.15.4, che era già stato sviluppato per il sistema operativo Erika EE, all’interno di un ambiente FreeRTOS. Il tutto viene eseguito in un contesto di Asymmetric Multi Processing, al fine di delegare la gestione di reti IoT real-time a processori dedicati, mentre quello principale può occuparsi della gestione di tutte le altre periferiche. Al giorno d’oggi uno dei settori tecnologici più interessanti è sicuramente l’Internet of Things. L’IoT ha come obiettivo la produzione di oggetti intelligenti, che siano in grado di interagire tra di loro all’interno di una rete. Una rete è composta da sensori e dispositivi, i quali possono anche essere connessi a Internet. L’attuale crescita del mondo IoT è resa possibile dalla costante evoluzione dell’elettronica, che riesce a produrre dispositivi potenti nonostante i bassi prezzi, le sempre più piccole dimensioni e il basso consumo energetico. All’interno dei sistemi embedded, l’inclusione di più di una CPU è ormai diventata una prassi, e questo fatto permette di migliorare notevolmente le performance dell’hardware. La tecnica dell’Asymmetric Multi Processing può essere usata per gestire la presenza di core multipli. Ciascun core possiede una propria architettura e ogni CPU può eseguire un’istanza di un diverso sistema operativo. Una possibile soluzione per sfruttare il potenziale dell’AMP è utilizzare un sistema operativo come Linux in un core, mentre sull’altro core gira un sistema operativo real-time. L’RTOS gestisce tutte quelle operazioni che richiedono una computazione entro una certa deadline, mentre l’altro sistema operativo gestisce le applicazioni di più alto livello. L’uso dell’AMP quindi è una soluzione ottima per un grande insieme di applicazioni IoT, in quanto permette di assicurare il rispetto dei vincoli real-time e allo stesso tempo di ottenere prestazioni elevate. In questa tesi un sistema dual-core (la scheda VAR-SOM-MX7) viene utilizzato in contesto di AMP. Una possibile applicazione IoT potrebbe essere la creazione di una rete wireless di sensori per la gestione della temperatura, dell’illuminazione o dell’umidità di un edificio. Per la gestione della rete, c’è bisogno di scegliere anche un appropriato protocollo di comunicazione che permette ai dispositivi di comunicare l’un l’altro. Dato che questo tipo di applicazione richiede di scambiare piccole quantità di dati a brevi distanze, il protocollo IEEE 802.15.4 potrebbe essere la scelta ottimale. Il dispositivo MRF24J40 è un dispositivo radio che implementa questo standard, e può essere collegato alla scheda hardware, che fungerà da coordinatore della rete. In questo lavoro, descriveremo il protocollo ed effettueremo il porting del driver del dispositivo e dello stack protocollare all’interno di un ambiente FreeRTOS, sfruttando la piattaforma hardware VAR-SOM-MX7, il cui processore accoppia un core ARM Cortex-A7 a un core ARM Cortex-M4.

Abstract

This thesis aims at porting the IEEE 802.15.4 protocol stack, which was already developed for the Erika EE operating system, within a FreeRTOS environment. This is done in a context of Asymmetric Multi Processing, in order to delegate the management of real-time IoT networks to dedicated processors, while the main core can manage all the other devices. Nowadays the Internet of Things is certainly one of the most interesting technological sectors. IoT aims to produce smart things which can interact with each other within a network. A network consists of sensors and devices, that can also be connected to the Internet. The current growth of the IoT world is made possible by the constant evolution of electronics, which produces powerful devices despite the low prices, the small sizes and the low power consumption. Within the embedded systems, the inclusion of more than one CPU is now a practice, and this fact allows to considerably improve hardware performances. Multiple cores can be managed with an Asymmetric Multi Processing technique. Each core has its own architecture and every CPU can execute an instance of a different operating system. A possible solution for exploiting the potential of AMP is to run an operating system like Linux in a core, while the other core runs a Real Time Operating System. The RTOS manages operations that require a computation within a deadline, while the other OS handles applications of higher-level. AMP therefore is an optimal solution for a large set of IoT applications, providing high performances and ensuring to respect real-time constraints. In this thesis a dual-core board, the VAR-SOM-MX7, is used in a context of Asymmetric Multi Processing. The processor of the board couples an ARM Cortex-A7 core and an ARM Cortex-M4 core. A possible IoT application could be the creation of a wireless sensor network for the management of the temperature, the lighting or the humidity of a building. For the network management, there is the need to choose an appropriate communication protocol which permits the devices to communicate each other. Since this kind of application requires to exchange small amounts of data over short distances, the IEEE 802.15.4 protocol could be an optimal choice. The MRF24J40 is a radio device that implements this standard, and it can be connected to the hardware board. The board can act as the network coordinator. In this work, we describe the protocol and we port the device driver and the protocol stack within a FreeRTOS environment, using the VAR-SOM-MX7 hardware platform.

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Internet of Things

Internet of Things (IoT) is an expression used to define every object that is connected to a network. A network is composed of sensors and devices, which can also join the Internet. This connectivity is possible thanks to a set of wireless protocols and modern hardware components. Nowadays IoT is one of the most interesting area of research and it is an ever-expanding sector. IoT aims to produce smart things and make them interact one another as well as with us. Any object that we use every day can be transformed in a smart device. IoT devices are realized to assure two functions: control and monitoring. Connectivity and communication technologies make possible, for a user, to monitor remotely the operation of an IoT device. Sensors and tags permit the object to relate with the surrounding environment, detect useful data and adapt accordingly. IoT uses sensors, devices and any other type of smart object to offer services to the applications, ensuring, at the same time, that security and privacy requirements are satisfied.

An IoT architecture is typically based on three elements: the things, the network and the cloud. The things include all the objects, devices or sensors that are connected to the network through a cable or wireless. The network connects the things to the cloud which is used to store data on a remote server without security problems. Data are generated by sensors and devices and, for example, they can represent information derived from temperature, humidity or position sensors. The data that, if necessary, are transmitted to the cloud through the network are traced and consolidated, becoming bigger and bigger over time. This huge amount of information allows to control a system in an optimal way.

The IoT term indicates all the technologies that allow to transform any object into a device connected to the internet. These technologies thus include also the communication protocols which permit the devices to communicate each other. A lot of protocols can be used within an IoT environment and the choice of which is the best solution is based on the application requirements. Factors that most influence the selection of the appropriate protocol are the needed data rate, power consumption, range and frequency.

It is not possible to give a standard definition of IoT, since the term IoT indicates a constantly evolving hi-tech world. Technology improvements will allow the connection of an ever-increasing number of smart devices. The IoT benefits are widely exploited in a large number of sectors. Smart cities can be defined as the cities of the future, they use IoT to manage waste disposal and to control the water distribution system. Embedded sensors and IoT systems are also used to reduce congestion, to minimize energy consumption and to secure the cities. Smart homes are designed to simplify and improve people lives. It is possible to install systems to manage the temperature, the lighting and any other type of electronic device such as a fridge or a television, improving efficiency and reducing consumption. An IoT system can also be used for intruder detection or other security purposes. IoT devices are very important within the field of healthcare since they allow to monitor people improving its independence and its quality of life. Wearable fitness devices are useful to control a sport activity and wearable devices include also smartwatches and smart glasses. Other application areas for IoT can be agriculture, smart industries and smart cars. These applications will soon affect our lives, making them more simple and comfortable.

The concept of Internet of Things was theorized for the first time in 1999 by Kevin Ashton, to describe a system that connects objects belonging to the physical world to the Internet. It allows to connect every kind of object and device one another. If a computer could know everything from things, acquiring data, we can classify and keep track of everything around us. The growing IoT development goes hand in hand with the technological progress. Electronics produces increasingly powerful devices, small in size, at lower prices and with low power consumption. Information technology advancement has led to the development of sophisticated algorithms and to the improvement of cloud services, allowing to collect, scan and extract information from a large amount of data.

When we decide to create an IoT system, we do not have a unique framework that can be used for every application. Each project requires careful analysis over the network that we want to build, in order to decide the most appropriate communication protocols, development hardware and operating systems. The choice of these system components is a crucial point to implement the better solution for our needs.

1.2 Asymmetric Multi Processing

In the IoT world the inclusion of more than one CPU within an embedded system is becoming a practice, because nowadays applications need higher performances than the past. This fact has brought to the development of multiple solutions, and each of them has its own pros and cons. The two primary options to manage multiple cores are the Symmetric Multi Processing (SMP) and the Asymmetric Multi Processing (AMP) techniques. In both system types is present more than one CPU, and different cores can communicate each other.

In the SMP, different cores have the same architecture and share at least a part of the memory space. Moreover, a single Operating System takes care of distributing the workload between single CPUs and of handling the resource sharing. SMP is preferable if an application needs a lot of power for an optimal execution, and a single CPU can’t guarantee a right management of the workload. The processors are symmetric, therefore each one can do any type of job.

In the AMP systems, each core can have its own architecture, the memory space is not shared and more Operating Systems can reside in memory. Each CPU can execute, if necessary, an instance of an Operating System. In the case different cores runs different OS, AMP is said to be heterogeneous. The sharing of the resources and the communication between cores is more complex than in SMP. This solution is particularly used when the architectures of the cores are perfect for specific activities. AMP is more difficult to implement, because individual processors are dedicated to specific tasks.

A possible solution for exploiting the potential of AMP mode is to run an OS like Linux in a core, while the other core runs a Real Time Operating System. The RTOS manages operations that require a computation within a deadline, while the other OS handles applications of higher-level. AMP therefore is an optimal solution for a large set of IoT applications, providing high performances and ensuring to respect real-time constraints.

The AMP could be used for the management of the coordinator node of a generic Wireless Sensor Network. As we illustrate in the following chapter, a large group of application areas use WSNs. In the home automation sector, sensors networks can be used to manage lighting, temperature, home entertainment, intruder detection and a lot of other things. These networks are spread also within industrial automation, agriculture and almost all the other IoT fields that require a wireless monitoring system. A sensor usually represents an end device of the network. These sensors transmit their data to a concentrator node, which act as the network coordinator.

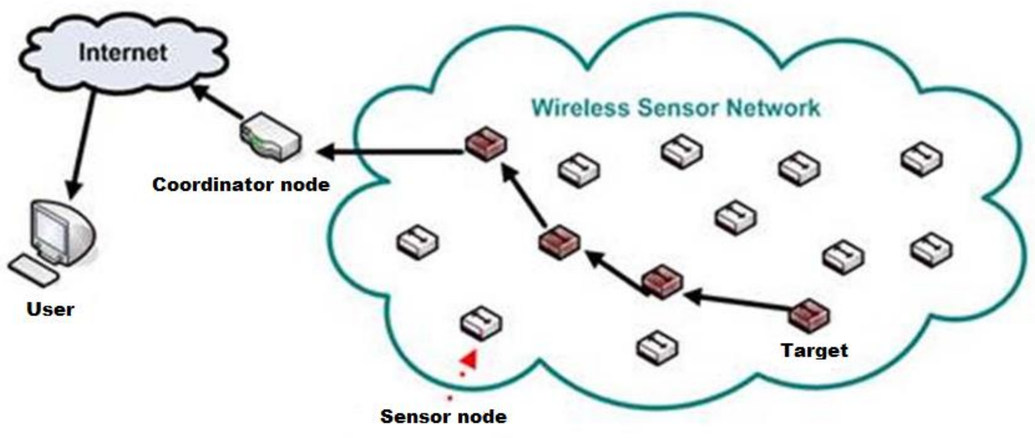


Figure 1.1: General Wireless Sensor Network structure.

The concentrator node could be a VAR-SOM-MX7 board, to which a radio transceiver is connected. This dual-core board that works in a context of Asymmetric Multi Processing couples an ARM Cortex-A7 core, which runs Linux, and an ARM Cortex-M4 core, equipped with FreeRTOS. A sensor network can transmit data to the M4 core of the coordinator, that runs a real time operating system. This possibility allows to manage the IoT sensor network respecting real time constraints. Data can be further transmitted to the A7 core, which in the meantime deals with more complex elaborations that do not require to comply with real-time deadlines, such as the management of other peripherals. If needed, all these information data are also send to the cloud, that can store them for future requests. The aim of the thesis is to port the radio transceiver driver and the IEEE 802.15.4 protocol stack within FreeRTOS.

Chapter 2 gives an overview of the IEEE 802.15.4 protocol, specifying why this standard is one of the most used within the Internet of Things. Chapter 3 presents the FreeRTOS real-time operating system and explains how this operating system is suitable for developing applications that respect real time constraints. Chapter 4 describes the development hardware which includes the VAR-SOM-MX7 board and the MRF24J40 radio transceiver. Chapter 5 is dedicated to the presentation of the developed code. We describe the procedure necessary to realize a FreeRTOS application that includes the IEEE 802.15.4 standard, focusing first on the integration of the transceiver into the hardware board, and subsequently on the porting of the radio transceiver driver and of the protocol stack within a FreeRTOS environment.

Chapter 2

Protocol description

IEEE 802.15.4 standard is a network protocol used in many IoT solutions for implementation of low-rate Wireless Personal Area Networks (WPANs). It was introduced as an alternative to other existing standards, among which the IEEE 802.15.1 protocol (Bluetooth), a low-power and low-rate wireless technology, excellent for point-to-point communications, and IEEE 802.15.3, utilized in high-rate Wireless Personal Area Networks. Since not all applications require high data-rates or wide areas of coverage, the 802.15.4 protocol was established to support all those applications that need to exchange small amounts of data over short distances. Considering that Bluetooth technology doesn't support multiple-node networks, a new standard was introduced to allow the development of applications that require low complexity, low power consumption, low data-rate, short communication range, easy installation and low cost. Low power consumption allows to use battery powered devices, in order to build a wireless network, reducing installation costs. The absence of cables facilitates installation, reduces costs, but also cuts down communication ranges.

The application areas of networks based on the IEEE 802.15.4 protocol vary widely inside of industrial and domestic sectors. This standard is the perfect solution if the reduction of costs and consumptions has a significant importance. IEEE 802.15.4 based networks are used for these purposes:

- Home automation and security: a wireless personal area network makes available a low-cost solution for home control, such as lighting, home entertainment, intruder detection, fire detection, doors control.
- Consumer products: WPANs can be used inside electronic products. These networks are useful for the control of computer systems, toys, home entertainment systems.
- Healthcare: a WPAN can be built with sensors and diagnostic devices, for example to monitor a fitness training or to manage a medical application.

- Vehicle monitoring: all vehicles are equipped with a lot of sensors, that can be organized in a wireless personal area network.
- Agriculture: sensors networks can assist farmers in their work, monitoring land and environmental conditions. These networks may have wide geographical coverage, but it is necessary to choose a proper and complex network topology.

In this chapter we introduce the OSI model, with a description of the seven communication levels that compose it. Starting from model explanation, it is simpler to understand which levels are implemented in the IEEE 802.15.4 protocol. In the second paragraph we describe the protocol, dedicating particular attention to its beacon-enabled version with Guaranteed Time Slots. This protocol may be used in many IoT applications, as will be later explained.

2.1 OSI model

The communication functions of a computing system are standardized by the Open Systems Interconnection (OSI) model, which offers a conceptual model, without knowing the internal structure and the technology used by the system. It is composed by a set of seven levels, called layers, which contain some related aspects concerning communication between two nodes of a network. The OSI model incapsulates messages coming from a given level in messages of the lower level. This structure permits to realize a multilevel communication, very useful because in this way is possible to choose the protocols suitable for the specific application. Below there is a brief description of each layer:

- *Physical layer*

  It must define physical details of the connection, it takes care of transmitting data bits from the sending device to the receiving device, through a communication medium, with an appropriate bit rate. At this level, data are transmitted by means of signals supported by the communication medium, such as radio frequencies or electric voltages. Transmission can be simplex, half duplex or full duplex.

- *Data Link layer*

  It offers a link between two nodes of the network and, if possible, it checks and corrects physical transmission errors. It also provides functionalities to establish and terminate a connection between two devices that are physically connected. IEEE 802 standard splits this layer in Medium Access Control sublayer and Logical Link Control sublayer. MAC layer controls how a device accesses to the communication medium and how it transmits data. LLC layer deals with recognizing network layer protocols, error examination and frame synchronization.
- *Network layer*

  This layer introduces the concept of routing. When a data frame reaches this layer, source and destination addresses are controlled, to verify if the data frame is arrived at the destination. If data are arrived at the destination device, the layer delivers the data packet to the upper layer, otherwise the destination address is updated, and the data frame passes to the lower layer. Network layer also achieves the mapping between logical and physical addresses.
- *Transport layer*

  It delivers data sequences from a source to a destination, through network connections. It supports a set of properties such as error recovery, flow control and retransmission of undelivered packages.
- *Session layer*

  It manages network connections and it also takes care of initialize and terminate a connection. It controls communication between computers.
- *Presentation layer*

  It is probably the simplest layer, it transforms data into a form that the application can accept. It establishes the framework between application layer entities.
- *Application layer*

  It is the layer that provides network services to the applications, it is very close to the end user. This layer can interact with the software application.

IEEE 802.15.4 standard defines only physical and MAC layers.

2.2 IEEE 802.15.4 protocol

The standard defines lower levels specifications of the ISO/OSI model (i.e. Physical layer and Medium Access Control layer). Other levels are implemented in a suitable manner for specific applications. ZigBee and MiWi are two examples of protocols based on the IEEE 802.15.4 standard. Typical characteristics of a low bit-rate network based on this protocol are:

- Bit-rate usually equal to or less than 250 kb/s
- Star or peer-to-peer topology
- Carrier Sense Multiple Access with Collision Avoidance for accessing the channel
- Optional Guaranteed Time Slots
- Optional acknowledgment after data transmission
- Low energy consumption

2.2.1 Network topology

The nodes that make up a IEEE 802.15.4-based network are subdivided into two types: full-function devices (FFD) and reduced-function devices (RFD). A FFD can work as network coordinator or as common node, and it is able to communicate with any other device. On the contrary, a RFD is a device that can communicate only with a FFD, and it can't become a coordinator. Usually RFDs are very simple devices, without many resources available, that handle limited data traffic. Therefore, in a network, the presence of at least a FFD node is necessary. It will play the role of network coordinator.

Each IEEE 802.15.4 wireless personal area network can be constructed with star or peer-to-peer topology, depending on the application. A network consists of at least one full-function device which acts as network coordinator, while all other devices, placed at appropriate distances, can be both FFD and RFD. Each device is associated with a 64-bit identifier, but also 16-bit identifiers are available in some limited environments.

In a star topology the central node is the coordinator of the network. This node deals with creating the personal area network, declares itself as network coordinator and waits until other devices decide to become part of the network. The first FFD node that is activated becomes the PAN coordinator. Each star network is independent of the others, and the PAN identifier is unique compared to all other networks in the communication range. The communication is only possible between a device and the central coordinator. The Personal Area Network coordinator generally is mains powered, while other nodes are principally battery powered. Applications that use star topologies are home automation systems, health-care devices, sensor networks.

Peer-to-peer networks are composed by devices connected in an arbitrary way. Their unique constraint is the restrained distance between a couple of nodes, which must not be exceeded. Even in these networks there must be a PAN node coordinator and all the devices can communicate one another. A node becomes the coordinator if it is the first that transmits on the selected channel. Usually these networks are able to self-manage the connection organization. Multi-hop communication is also possible, increasing network complexity, but a support for this type of routing operations must be provided by an additional network layer, not defined in the standard.

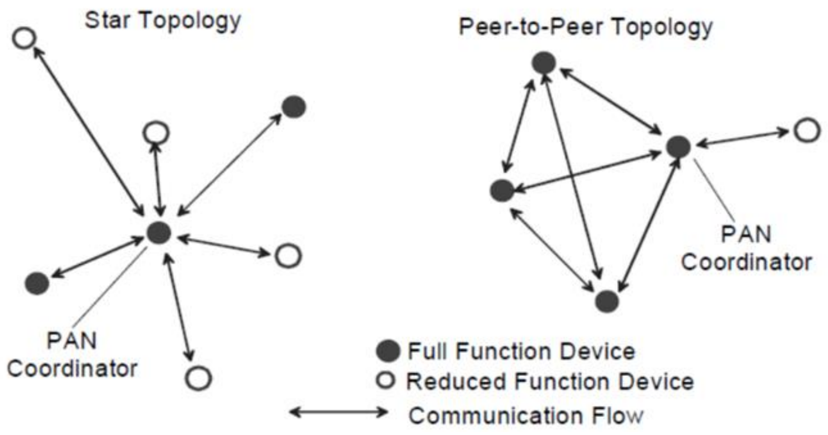


Figure 2.1: Star and Peer-to-Peer network topologies.

Cluster-tree networks are peer-to-peer networks organized in a structure in which each reduced-function device is associated with only one FFD, RFDs correspond to the leaves of the tree, while most of the nodes are full-function devices. In this case multiple networks can be joined, they make up a more complex network which permits communication between distant nodes. Each subnet has a coordinator which takes care of routing operations.

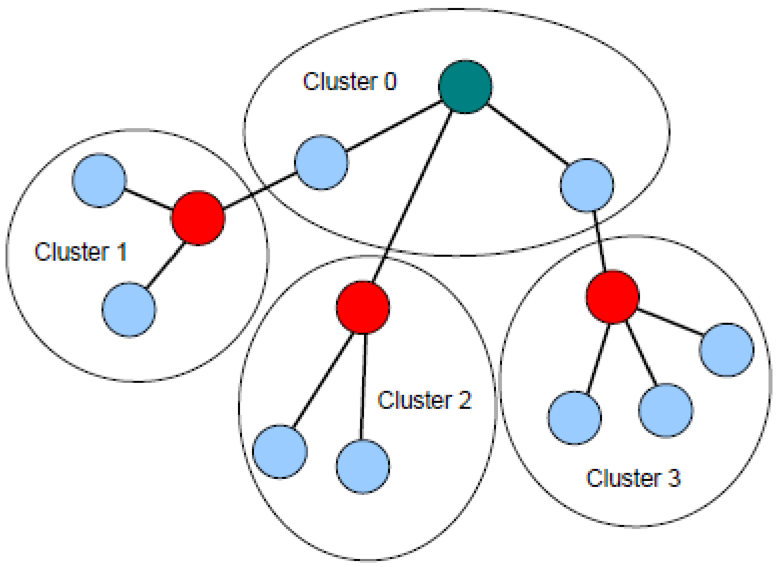


Figure 2.2: Cluster-tree network topology.

2.2.2 Frames

IEEE 802.15.4 data are organized into packets. Packets, also called PHY protocol data units (PPDU), are transmitted through the physical channel. Packet structures have been studied to ensure robustness for transmission in presence of noise, while keeping the complexity very low. PHY packets include a synchronization header (SHR), a PHY layer header (PHR) and a payload (PSDU). SHR contains a preamble field and a Start of Frame Delimiter field (SFD), that indicates the end of the SHR and the start of the packet data. PHR specifies the packet length, while PSDU represents the physical layer payload. PSDU, which equals to the MAC frame, comprises a MAC header (MHR), a MAC payload and a MAC footer (MFR). Every PSDU (Physical layer Service Data Unit) is controlled through CRC (Cyclic Redundancy Check).

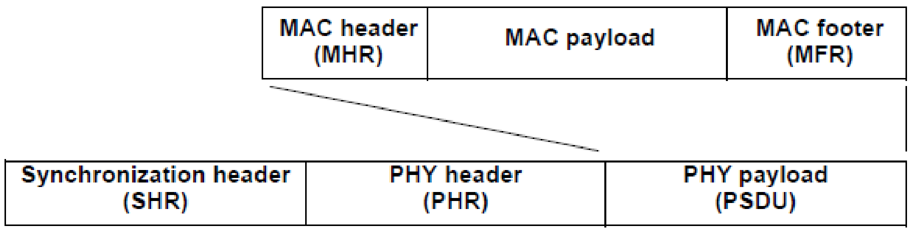


Figure 2.3: Schematic view of the PPDU.

In the IEEE 802.15.4 standard, basic data transport packets are called frames, which can be divided into four categories:

- Data frames
- Acknowledgement frames
- Beacon frames
- MAC command frames.

The structure of these frames is described below.

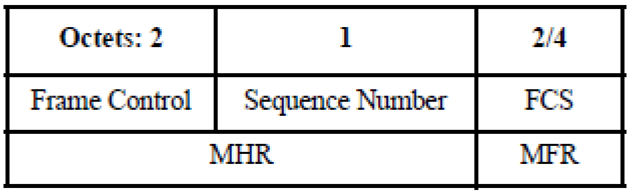
The MAC header includes a Frame Control field, that contains packet specific parameters, a Sequence Number, sender and recipient addresses and an Auxiliary Secondary Header, which holds security information.

A data frame is a generic frame used to transfer every kind of data, the payload contains data exchanged between the nodes.

Octets: 2	0/1	variable	variable	variable		variable	2/4
Frame Control	Sequence Number	Addressing fields	Auxiliary Security Header	IEs		Data Payload	FCS
				Header IEs	Payload IEs		
MHR					MAC Payload		MFR

Figure 2.4: Data frame structure.

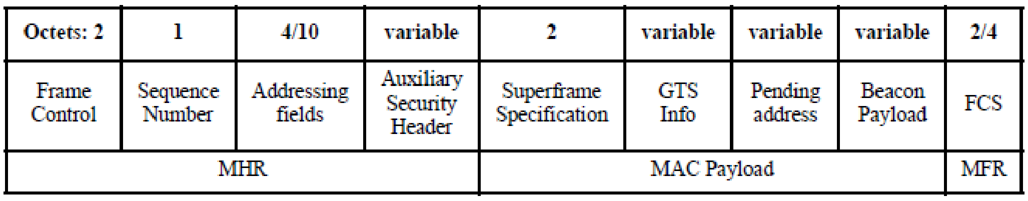
An ACK frame is used by the receiver to confirm the success of a data transmission. The Sequence Number equals the SN of the received frame. The ACK is transmitted to the data frame sender.



Octets: 2	1	2/4
Frame Control	Sequence Number	FCS
MHR		MFR

Figure 2.5: ACK frame structure.

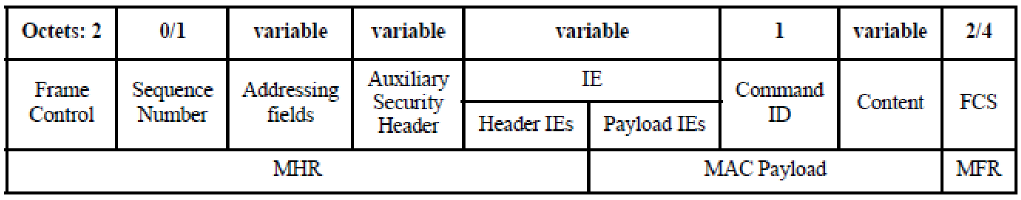
A beacon frame is used by coordinators to transmit a beacon. Two beacons delimit a superframe. A beacon contains information regarding frame duration and Guaranteed Time Slots.



Octets: 2	1	4/10	variable	2	variable	variable	variable	2/4
Frame Control	Sequence Number	Addressing fields	Auxiliary Security Header	Superframe Specification	GTS Info	Pending address	Beacon Payload	FCS
MHR				MAC Payload				MFR

Figure 2.6: Beacon frame structure.

A MAC command frame is used to manage network operations, as association requests, data requests or Guaranteed Time Slot requests.



Octets: 2	0/1	variable	variable	variable		1	variable	2/4
Frame Control	Sequence Number	Addressing fields	Auxiliary Security Header	IE		Command ID	Content	FCS
				Header IEs	Payload IEs			
MHR					MAC Payload			MFR

Figure 2.7: MAC command frame structure.

2.2.3 Superframe structure

The standard includes the possibility of exploiting a superframe structure, with which packets can be transmitted only during certain time slots. A superframe is delimited by two beacon signals generated by the network coordinator. Beacons are used to coordinate the connected nodes, to describe the superframe and to identify the network. A superframe can be divided in two sections, a Contention Access Period (CAP), followed by a Contention Free Period (CFP), in which timeslots are reserved for certain nodes. A superframe typically is composed by 16 timeslots and is followed by an inactive period, during which the coordinator switches to sleep mode, saving energy. Contention Access Period represents the first part of the superframe, it starts after the transmission of a beacon and it terminates before the Contention Free Period. If there isn't a CFP, CAP finishes at the end of the superframe active section. All the frames that are transmitted during the Contention Access Period use a slotted CSMA/CA algorithm to access the channel. Contention Free Period starts at the end of the Contention Access Period. If some Guaranteed Time Slots have been allocated by the coordinator, they occupy contiguous slots of this section. The access to the channel isn't managed by a CSMA/CA algorithm, since timeslots are reserved.

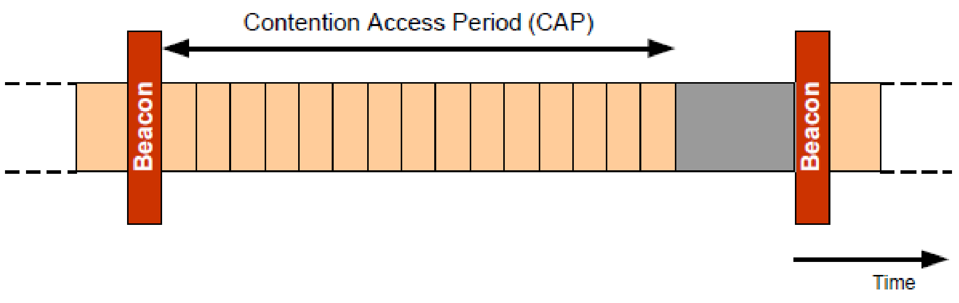


Figure 2.8: Superframe without Contention Free Period.

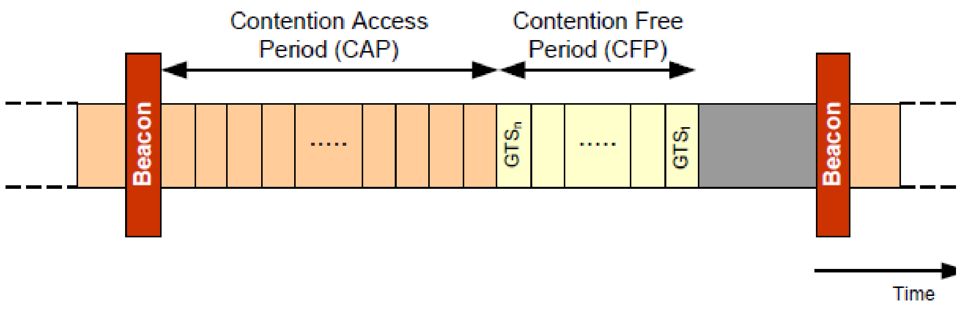


Figure 2.9: Superframe with a Contention Free Period.

2.2.4 Beacon-disabled and beacon-enabled modes

IEEE 802.15.4-based networks always use beacons when a new device joins the network. Beacons are signals generated by the coordinator. A network based on this protocol, can work in a beacon-disabled or in a beacon-enabled mode.

In beacon-disabled mode, beacons aren't regularly transmitted by the network coordinator. A general device communicates with the coordinator solely if it needs to. In this way power consumption is limited, because communications can be rare. A node, to establish if there are pending data for it, must poll the coordinator. This mode is an optimal solution when the coordinator and the other nodes exchange few data among them, generating light traffic.

In beacon-enabled mode, devices synchronize to the network thanks to information contained in beacon signals, that are periodically produced by the coordinator. Beacons also contain information on the data frames pending for any other network node. Usually a superframe is bordered by two successive beacons, and it provides sixteen timeslots that network nodes use to communicate.

2.2.5 Guaranteed Time Slots

Network nodes can request the exclusive assignment of superframe timeslots. These timeslots, the Guaranteed Time Slots, form the Contention Free Period, located after the Contention Access Period of a superframe. During the CFP, communication doesn't need a CSMA/CA mechanism for channel access.

Guaranteed Time Slots (GTS) are very useful because they allow to assign portions of a superframe to a device. The device, during the time slot, has exclusive access to the channel and it can communicate with the network coordinator. There is the possibility to allocate up to 7 Guaranteed Time Slots. GTSs allocation and deallocation are responsibility of the coordinator, while other devices make allocation requests. Deallocation can be made by the coordinator or by the device that possess the slot. A Guaranteed Time Slots has a transmission direction, so a device may request two GTSs: a transmit GTS and a receive GTS. GTS information, such as the result of an allocation, is transmitted in a beacon.

2.2.6 Data transfer

IEEE 802.15.4 standard includes a set of transfer opportunities:

- A device transmits data to the coordinator
- A device receives data from the coordinator
- Two peer devices exchange data.

If the correct reception of a data frame requires an acknowledgement, the device that receives the frame is responsible for answering with an appropriate acknowledgement frame.

In a beacon-disabled personal area network, if a device needs to transfer data, it sends the data frame to the coordinator. In beacon-enabled mode, when the device wants to send data to the network coordinator, it waits a beacon. When the device finds the beacon, it synchronizes to the structure of the superframe and, when it gets the permission, it transmits the data frame to the coordinator.

Conversely, when the coordinator has to transfer data to a device, if beacons are enabled, it signals in the pending list of the beacon that there is a ready data frame. The device periodically listens the beacon and, when it finds a pending message, it sends a data request command. At this point, the coordinator transmits the data message and, when the transmission is completed, the message is removed from the pending data frame list. In a beacon-disabled personal area network, a device requests data to the coordinator by sending a data request command frame. The coordinator, if a data message is pending, transmits the frame. Otherwise, in case there are not pending messages, the coordinator sends a data frame with void payload or, if an acknowledgement was requested by the device, it transmits an acknowledgement frame which explains that the pending list is empty.

In a peer-to-peer topology, a device can receive data from any other device. When a device gets access to the channel, it can transmit the data frame.

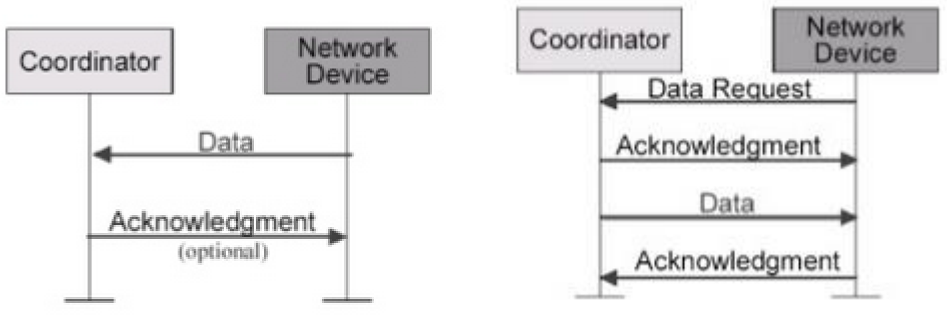


Figure 2.10: Beacon-disabled data transmission scheme.

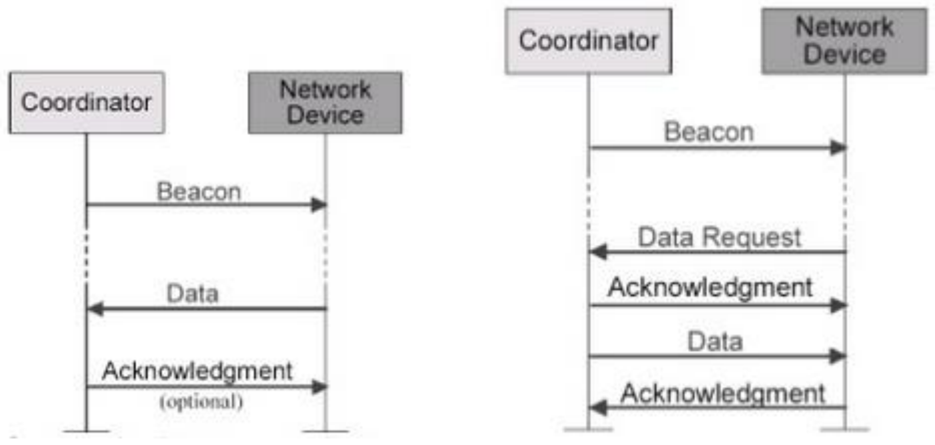


Figure 2.11: Beacon-enabled data transmission scheme.

When a device wants to enter in a personal area network, if the coordinator is available to accept new nodes, it must send an association request frame to the coordinator. The coordinator answers with an acknowledgement and communicates its decision. If the coordinator accepts the device, it assigns to the device a new short address. The device finally sends an ACK to the coordinator.

A disconnection can take place if the device wants to leave the network, or if the coordinator needs to disconnect the device from the network. In the first case the device sends a disassociation notification to the coordinator, and it waits an acknowledgment from the coordinator. In the other case the coordinator sends a disassociation notification to the device. The device responds with an ACK.

2.2.7 Security mechanism

Obviously in wireless networks there is not the necessity to take control of the wire to communicate and for this reason WPANs can be attacked through eavesdropping and tampering. Low rate Wireless Network devices are often low-cost, with limited available memory space and limited computational capability, therefore it is not easy to secure these networks. WPANs usually have not a static infrastructure and a communication may occur between two devices that previously were not interconnected. Embedded systems that use this standard also require limited cost and battery consumption, so the security architecture must be carefully managed, and the cryptographic algorithms can’t be too much complex.

The standard uses a symmetric-key cryptographic system, with keys that are securely delivered by higher layers. This mechanism assures that transmitted data is solely revealed to the recipient to whom it is addressed, and it assures that the source of information is recognized. Furthermore, data can’t be altered during transmission and there must be a detection of duplicate information. Frame protection allows to exploit a key shared between two devices or alternatively, a key shared between a set of devices. In these two different cases, the level of protection isn’t the same but, using a group key, storage and maintenance costs are reduced. In a peer-to-peer system, group keys don’t offer protection against a malicious device located within the group, but only against an external attack.

Chapter 3

FreeRTOS

A generic operating system is a software that offers facilities to computer applications, performing a set of basic and indispensable operations. An operating system can seemingly execute more than one program at any time. It is necessary to give the user an impression of real-time responsiveness. A processor, if consists of a unique core, as a matter of fact executes only a task at a time. The core portion of an operating system is the kernel, a software that assures to the processes running on the computer a secure and controlled access to the hardware. The kernel manages the use of the CPU, distributing processor time between programs. A kernel component, the scheduler, is dedicated to this purpose. The scheduler of a Real Time Operating System must guarantee a deterministic execution scheme. RTOS are particularly used in embedded systems, which frequently require the respect of real time deadlines.

FreeRTOS is a real time operating system suitable for developing real time applications within an embedded environment. Specifically, this real time kernel ensures to respect hard real time constraints, namely the time deadlines requested by the application must be guaranteed, otherwise the system fails. FreeRTOS is one of the most used real-time operating systems. The success is due to its simplicity, indeed the source code is almost exclusively written in C.

In this chapter we describe the structure of a FreeRTOS distribution, particularly focusing on the kernel source files. Afterwards we introduce multitasking, and we explain how FreeRTOS manages tasks, mutexes, memory and timers.

3.1 Kernel source structure

A microcontroller is a resource limited system that includes a processor, ROM to store a program and RAM required to execute the program, all within a single chip. These hardware architectures are specifically built for embedded applications, which exploit restricted resources to execute a certain job. Since the available resources are not unlimited, often it is not possible to use a full implementation of a real time operating system. For this reason, FreeRTOS includes only inter-task communication, timing and synchronisation primitives, beyond that the real time scheduler. It is indeed depicted as a real time kernel and every other needed component is added with an add-on module, as it will be done for the IEEE 802.15.4 networking stack.

FreeRTOS Application Program Interface (API) wants to be as much as possible simple and easy to use. This operating system supports many ports (a port is intended as a microcontroller architecture coupled with a compiler) and, for each port, FreeRTOS provides an official demo, which can be executed in the appropriate hardware. These demos are made available to simplify the first FreeRTOS approach to the user.

Any FreeRTOS project, if possible, is created by starting from one of these demos, to guarantee the application includes all the files (both source and header files) necessary for a correct execution. Starting from a demo, also ISRs are installed without any developer effort. A FreeRTOS application is very similar to any other not real-time application. The substantial difference is the call to the function *vTaskStartScheduler()*, which is called within the *main()* function. It starts the scheduler of the real-time operating system. After this call, the RTOS starts to manage tasks execution.

3.1.1 Source files

FreeRTOS source files, that are written in C, are built together with the other C files of the application. In every project, beyond that source and header files, there must be a *FreeRTOSConfig.h* file. This file is specific for each application, it adapts the kernel of the operating system to the project, and it is situated inside of an application directory. Within this file the application developer customizes the RTOS, depending on what he needs. For example, the *configTOTAL_HEAP_SIZE* definition configures the heap dimension, while *configMINIMAL_STACK_SIZE* defines the minimum stack dimension used by a task. The idle task will have this size.

A portion of FreeRTOS source files is common to all ports, while other files are distinctive of each couple compiler - processor. FreeRTOS code is organized in a set of subdirectories:

- ***doc:*** This folder contains all FreeRTOS documentation files.
- ***examples:*** It includes demo examples, specific for the VAR-SOM-IMX7 board.
- ***middleware:*** It contains Open Asymmetric Multi Processing Framework, a set of software components useful for developing applications in AMP context. For example, this directory includes the RPMsg API that permits Inter Processor Communication (IPC) between autonomous software running on different cores.
- ***platform:*** This folder includes platform specific drivers and utilities.
- ***rtos:*** This is the kernel source code of the operating system.
- ***tools:*** The directory comprises the cmake toolchain files.

The core of the FreeRTOS source code is include within the *rtos* directory. The structure of the source code is here described:

- ***include*** This folder contains all the include files of the RTOS.
- ***portable*** This folder includes a *MemMang* directory which contains heap memory allocation schemes, and other directories that contains files specific for a particular microcontroller or compiler.
- ***croutine.c*** It implements all co-routine functionalities. This file is included only if co-routines are used into the application. Co-routines were created to support small devices, but they are rarely exploited in modern applications, so their code is not supported as other FreeRTOS code. They are similar to tasks, but they share a unique stack. They use a cooperative scheduling algorithm, and, because of limited available resources, they have a lot of limits regarding their

implementation structure. Tasks are always preferred to co-routines.

- ***event_groups.c*** It provides event group services and it is included in the project only if event groups are needed. An event group is a sequence of bit to which the application assigns a meaning. A task can examine the bit values and thanks to them, it understands which events are active.
- ***list.c*** It includes a list implementation, this solution is specifically built for scheduler needs, but it can be also used within an application. It is always included within a project. The list is composed by a group of list items. Each item comprises a numeric value that is utilized to sort the list in a descending way. When a list is created, it contains an item, coupled with the maximum numeric value. This item works as an indicator and it is located always at the end of the list. Each list item includes a pointer to the next list item and a pointer to the list in which it is contained.
- ***queue.c*** It provides queue and semaphore services to a FreeRTOS application. This file is often necessary within the project. Queues are a basic type of communication between tasks, they are used to exchange messages also between tasks and interrupts. They typically are employed in form of thread safe FIFO buffers. This implies that new data are sent to the back of the queue and are extracted from the front.
- ***tasks.c*** This file supplies task facilities, it provides a set of API functions that the application developer uses to manage tasks. This file is included in every FreeRTOS project. Tasks are better descripted in section 3.2.
- ***timers.c*** This source file includes a set of software timer API functions. It is included within the application only if timers are going to be used.

3.2 Multitasking

In FreeRTOS, programs are called tasks and, considering that this operating system can run a lot of tasks concurrently, it can be defined as a multitasking operating system. The benefits of multitasking are numerous. A complex application can be divided in multiple tasks, which are smaller, easier to handle and they can communicate each other. Multitasking reduces the complexity of testing the software and distributing work among developers. It also allows to implement some features (as timing) inside of the operating system, simplifying the application code. If the processor of a microcontroller includes a unique core, it executes only a task at a time. Thanks to multitasking, from the user perspective, tasks are executed concurrently, because they are quickly switched, fully exploiting the processor time. The diagram below explains how the user perceives the task execution and how in reality the processor distributes time between tasks.

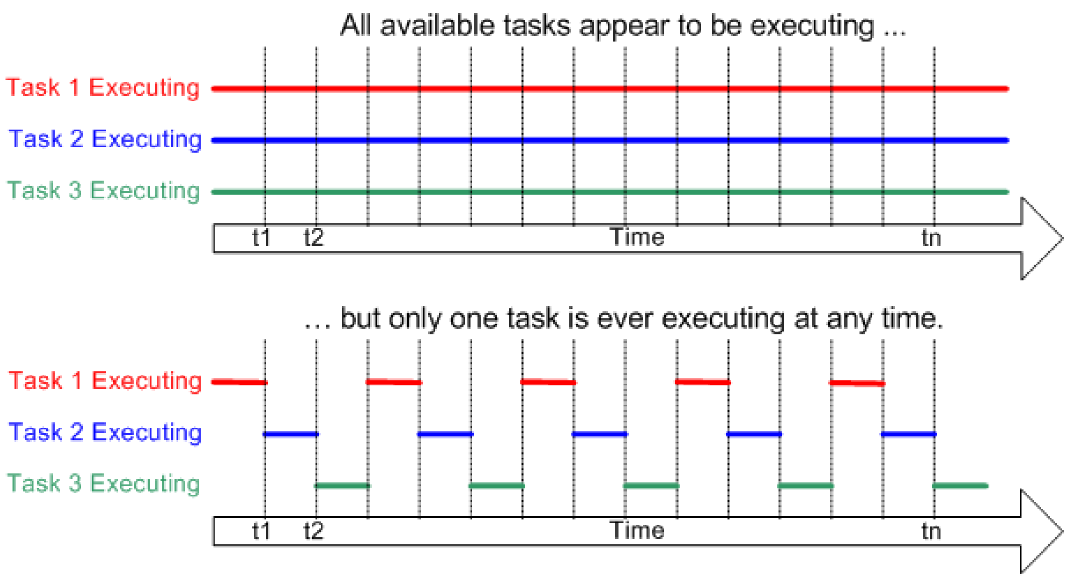


Figure 3.1: Time diagram of task execution.

The decision of which task has to be executed during each time interval is taken by the scheduler, an important piece of the kernel. Any task can be suspended and resumed many times, according to the policy adopted by the scheduler. This policy is implemented through a specific algorithm, which varies depending on the operating system and on the needs of the applications running on the system. In a not real-time system usually the processor time is equally divided among tasks, in a so called “fair” manner. Each task can be suspended by the scheduler when it terminates its time slice, but it can also decide to suspend itself. A task is suspended if it needs to delay for a certain time, if it wants a resource but it has to wait its availability or if it waits an event to happen. When a task is suspended, the scheduler cannot assign to it processor time, it can be executed only after awakening.

When a task is executing, it has its own execution context, that includes the registers of the microcontroller, beyond that RAM and ROM memory areas. Since a task is a simple piece of code, it cannot predict when it will be suspended and resumed by the scheduler. To prevent errors over data consistency, the kernel must ensure that the resumed task owns a context equal to the context that it had before the suspension. For this reason, the kernel saves the context of a task when it is suspended, while the context is restored before the task is resumed. This mechanism is called context switching.

Real time applications use multitasking for their activities, but they have different aims than not real-time applications. A crucial point is the scheduling policy. A real time system needs to respect real time constraints, since it has to assure a quick response to real world events. The scheduler guarantees deadlines are met. Each task has a priority and the scheduling policy implies that the processor is assigned to the ready task with the highest priority.

3.2.1 Tasks

Any real time application can be built as a set of stand-alone tasks and each task executes inside its own context. In every instant only one task can be in the running state and the decision of which task should be executed is responsibility of the RTOS scheduler. During the application execution a task can be swapped in and out (started and stopped) many times. Since each task holds its own stack, the processor context is saved to the stack whenever a task is swapped out and is restored when the task is swapped in. The context is composed by stack contents, register values and other information. The scheduler had to guarantee the proper functioning of this mechanism.

In FreeRTOS, tasks are implemented as C functions, with a *void* pointer passed as parameter and a *void* return type. A task can be simply considered as a small program with an entry point, which run forever within an infinite loop, without exiting, until the task is deleted (there isn't a *return* statement). This is the prototype of a task function:

```
void TaskFunction (void *Parameters);
```

3.2.2 States

When a processor includes a single core, at any time only one task can be in execution. A simple distinction can be made between *running* and *not running* states. If a task is running, the processor is executing its code, and all other tasks are not running. These tasks are waiting to be resumed by the scheduler, they had saved their contexts in the stacks and when they become active again, their execution flow starts from the instruction that was about to be executed before the suspension.

Not running state is further subdivided in ready, suspended and blocked states. A task can thus be:

- ***running***
  It is executing, using the processor. If the processor is composed by a single core, only one task at a time can be in this state.
- ***ready***
  It is not executing because another task is occupying the processor (a higher priority task) but it is ready to execute. It waits until the scheduler assigns the processor to it.
- ***suspended***
  It is not executing and cannot enter in the running state. A task enters and exits from this state when it is constrained through specific calls. The suspension occurs with the *vTaskSuspend()* call, while with *xTaskResume()* the task returns in the ready state, waiting the processor.
- ***blocked***

It is not executing and cannot be selected by the scheduler to enter in the running state. It is waiting for an external or a temporal event. It can wait for a semaphore event, or while a delay period has expired, for example.

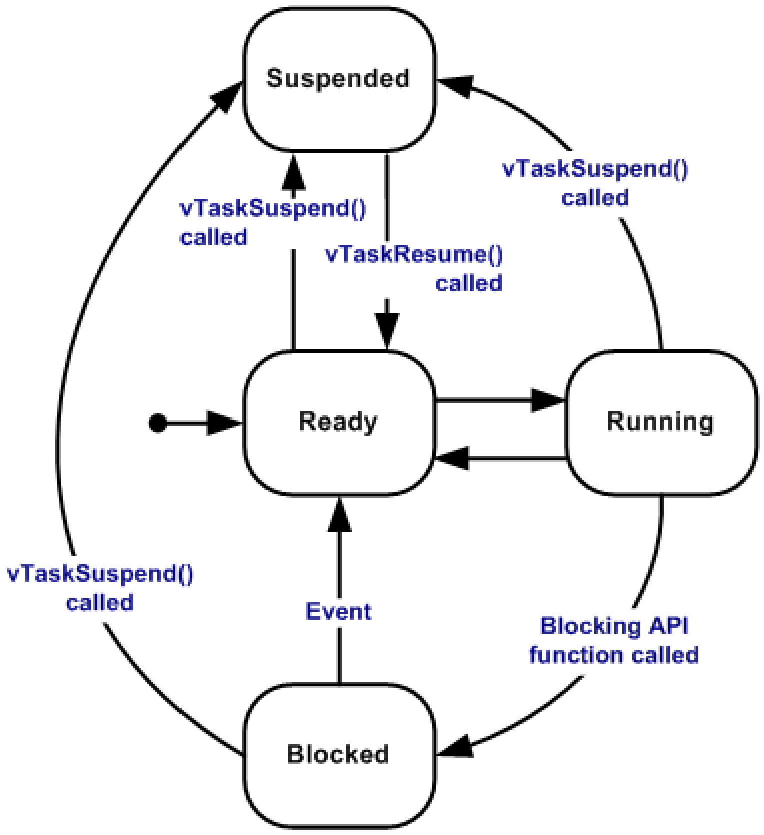


Figure 3.2: State transition scheme.

3.2.3 Priorities

When a task is created it assumes an initial priority, denoted by an integer, that can be changed with a specific function call after the scheduler has been started. Low priority values denote low priority tasks, and vice versa. Minimum priority is zero, which corresponds to the idle task priority. The idle task is created automatically by the operative system when the scheduler starts, to guarantee that there is always a ready task able to run. Its priority is set to zero because in this way, if there are other tasks in the ready state, he does not occupy processor time. This task is also responsible for freeing memory allocated by the operating system to tasks that have been deleted.

The scheduler will ensure that the higher priority task that is able to run (ready or running state) is preferred to other tasks of a lower priority, and that task is selected to enter in the running state, taking control of the CPU, till the scheduler will intervene again. Two or more tasks can share the same priority value. In this case, the solution is the adoption of a time sliced round robin scheduling policy. In other words, a task assumes the control of the processor for a determined amount of time (time slice), after which another task of equal priority enters in the running state, replacing the previous task, and remains running for an equal time slice, unless it suspends or blocks itself for other reasons. The priority of a task can be changed by means of the *vTaskPrioritySet()* function. The maximum value of the task priority is indicated in the configuration constant *configMAX_PRIORITIES,* so a priority can vary from 0 (the lowest priority) to configMAX_PRIORITIES-1 (the highest priority).

3.2.4 Task creation

A task is created through a call to the *xTaskCreate*() function (below, the function prototype).

```
BaseType_t xTaskCreate (TaskFunction_t pvTaskCode, const
char* const pcName, uint16_t usStackDepth, void
*pvParameters, UBaseType_t uxPriority, TaskHandle_t
*pxCreatedTask);
```

This API function returns a pdPASS or a pdFAIL value. The first value reveals the successfully task creation, while pdFAIL signals that the task has not be created, probably because there is not enough heap memory available to assign sufficient RAM. RAM is necessary to contain the task stack and other structures.

In the following table we describe *xTaskCreate()* parameters:

pvTaskCode	Like said before, tasks are functions written with C language and *pvTaskCode* parameter is a pointer to the function that realizes the task. The function always implements an infinite loop.
pcName	It is a human readable name, used for debugging. It usually describes the task, but it is not utilized by the Operating

	System.
usStackDepth	When a task is created, the kernel assigns to it a stack and *usStackDepth* indicates the breadth of the stack. This parameter requires a value that specifies the number of words the stack can store.
pvParameters	This pointer to void parameter represents the value passed into the task function.
uxPriority	It indicates the priority of the task. Priority values are assigned within a range that varies from 0 to (configMAX_PRIORITIES - 1).
pxCreatedTask	It is a handle to the created task. If the application does not need to handle the task, this parameter can assume a NULL value. It is needed to reference the task in future function calls.

Table 3.1: xTaskCreate() API function parameters.

3.3 Mutexes

Working with tasks, resource management is a crucial theme. Errors may occur when a task exits from the running state while it holds a resource, before concluding its work. Another task could need to access the same resource and it may cause problems like data corruption. Some examples of actions that can produce this type of errors are peripherals accessing, read, modify and write operations and non-atomic access to variables.

Mutual exclusion is a practice that guarantees data consistency when more tasks need to access to a specific resource. If the resource is shared between two tasks, the task that takes control of the resource gains an exclusive use, till it frees the resource. Even if the best solution to ensure mutual exclusion is to create an application that does not share any resource between more tasks, FreeRTOS makes available a set of features to realize mutual exclusion.

3.3.1 Critical sections

A critical section is a piece of code enclosed between two macro calls, *taskENTER_CRITICAL()* and *taskEXIT_CRITICAL().* Critical sections represent a first possibility for the implementation of mutual exclusion. Within a critical section, interrupts are completely or partially (up to configMAX_SYSCALL_INTERRUPT_PRIORITY) disabled, and context switches to other tasks are not permitted. If all the interrupts are disabled, the task which enters in the critical region, remains in the running state until it exits from this section. Critical sections generally are very brief because they must not influence the response time of an interrupt, and more critical sections can be nested, since the kernel is able to manage nesting. Calling a critical region macro is the unique solution that allows to modify the interrupt enable state of a processor. Critical regions are useful when the section of code to protect is very short.

A critical section can also be generated by means of scheduler suspension. In this case, interrupts are enabled, while other tasks can not enter in the running state until the section terminates. This solution is preferable when the section of code to be protected is long. The suspension of the scheduler is made through a *vTaskSuspendAll()* call. Interrupts are yet enabled, while context switches are disabled. Even context switch requests made by an interrupt must remain pending, until the scheduler is resumed. *xTaskResumeAll()* API call resumes the suspended scheduler. This function returns pdTRUE if a context switch that was pending is executed before the return statement, otherwise the returned value is pdFALSE. Below there are the prototypes of the two API functions.

```
void vTaskSuspendAll(void);
BaseType_t xTaskResumeAll(void);
```

3.3.2 Mutex semaphores

Mutexes are binary semaphores exploited to manage resources that are shared between multiple tasks. The main difference between a FreeRTOS mutex and a simple binary semaphore is the presence, in mutexes, of a priority inheritance system. They are available in a FreeRTOS applications only if the *configUSE_MUTEXES* parameter is set to 1 in FreeRTOSConfig.h. When two tasks share a resource, a mutex is linked to the resource. A task gains the access to the resource as soon as it takes the mutex and, once it finishes to use the resource, it must give the mutex back. After that the mutex returns free, another task can take control of it for accessing the resource.

FreeRTOS provides a sequence of API functions that permit to implement mutual exclusion within an application. In FreeRTOS, mutexes are a kind of semaphore and they are created with a *xSemaphoreCreateMutex()* function call:

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);
```

After being created the semaphore is stored in a variable of type *SemaphoreHandle_t,* that allows to manage the semaphore. If the function returns a NULL value, the semaphore can not be created, because of a lack of heap memory. The mutex semaphore can be taken and given back through *xSemaphoreTake()* and *xSemaphoreGive()* calls. The first function, used to acquire the control of the semaphore, returns pdTRUE value if the semaphore is correctly obtained, otherwise it returns pdFALSE, and requires two parameters:

- SemaphoreHandle_t xSemaphore: The semaphore handler.
- TickType_t xTicksToWait: The time (in ticks) to wait the availability of xSemaphore. If the time expires, pdFALSE is returned.

The other function, *xSemaphoreGive,* returns pdTRUE if the semaphore is properly released, pdFALSE if any error occurs. This function requires only an input parameter, the handle to the semaphore.
Lastly the API function *vSemaphoreDelete,* whose prototype is:

```
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);
```

deletes the semaphore passed as parameter. If a task is blocked because it is waiting the availability of the semaphore, this one cannot be deleted.

3.3.3 Priority inversion

Using mutexes, a problem called priority inversion may occur. We describe this case with an example.

Consider three tasks (Task1, Task2 and Task3). Each of them will have its own priority (Prio1, Prio2, Prio3). Suppose that:

Prio1 > Prio2 > Prio3.

Task1 and Task3 need to access to a resource through a mutex called MutexSem. If the lower priority task, Task3, takes the mutex, it can use the resource for an undetermined amount of time. When the higher priority task, Task1, tries to access the mutex, it blocks itself until the resource is released, despite of its higher priority. In the meantime, before Task3 gives back the mutex, Task2 wants to start. It does not need the resource and, thanks to its priority Prio2 > Prio3, Task3 is suspended and Task2 becomes running. In this case, Task1, which possesses the highest priority, has to wait for the end of both tasks Task2 and Task3. This undesirable situation is called priority inversion.

To avoid priority inversion, a mechanism called priority inheritance was introduced. This procedure increases the priority of the task that controls the mutex to the priority of the highest priority task that needs to access the same resource. In this way, the task inherits the priority of another higher priority task. Once the task releases the mutex, it returns to its initial priority.

3.3.4 Deadlock

An additional problem that may happen while using mutual exclusion is deadlock. Deadlock arises if two tasks are both suspended because they are waiting for a resource locked by the other task.

Suppose that Task1 and Task2 need to gain control of the same resources (ResA and ResB), through MutexA and MutexB. Task1 is running and takes the mutex of the resource ResA. Then Task2, which has a higher priority, becomes running, suspending Task1, and takes MutexB. After that, it tries to access resource ResA, that is not available, because this resource is locked by Task1. Task2 is blocked, so Task1 resumes its execution and attempts to access resource ResB. But ResB is not available, it is locked by Task2. At this point, Task1 enters in the blocked state, and deadlock occurs because both tasks are blocked. In the embedded systems, application designers are able to detect and remove sections where a deadlock may happen. In more complex systems there is a set of procedures that guarantee deadlock avoidance.

3.3.5 Task scheduling

When a mutex is shared between two tasks with different priorities, if both tasks must be executed the scheduler chooses the highest priority task as the next task that enters in the running state. If a task is blocked, waiting for a resource that is detained by a lower priority task, as soon as the resource is released, the low priority task is pre-empted, while the high priority task enters in the running state and takes the mutex. Conversely, when two tasks have the same priority, if a task finishes to use a resource, it exits from the running state only if it has terminated all its jobs, or if its time slice is expired. The other task that needs to use the same resource, awaits in the ready state.

3.4 Memory and time management

3.4.1 Memory allocation

Every time a task, timer or mutex is created, the kernel allocates a portion of RAM. This memory can be supplied by the developer of the application or, when an API function creates an object, the RAM can be dynamically allocated from the heap memory. In the case of dynamic allocation, *malloc()* and *free()* functions of the standard C library can be exploited. These functions however are not thread safe, they cannot be used in every embedded system, and the time needed to execute each function is not equal in every call, that is they are not deterministic. For this reason, a different memory allocation solution is necessary. Since different embedded systems may need different memory allocation methods, FreeRTOS provides five heap management systems. This memory allocation API functions are placed within the portable layer, outside the core source files. Each heap scheme can be used into an application, they implement various features and the application writer can supply its own heap management scheme. The feature of every scheme is now described:

- *Heap 1*
  This heap allocation scheme is the simplest implementation and it does not allow to free the memory, if it was previously allocated. This is use in a large number of applications, especially if objects are created when the system boot and they are never deleted during the program execution.
- *Heap 2*
  It authorizes to free previously allocated memory areas, but it does not merge contiguous free blocks. It is particularly used when the application needs to frequently delete objects. It can cause fragmentation if the memory allocated is of random size.
- *Heap 3*
  It modifies *malloc()* and *free()* standard C functions to assure thread safety.
- *Heap 4*
  This allocation scheme permits to merge contiguous free blocks in a larger memory area. It is useful to avoid fragmentation, contrary to what happens with heap 2 scheme. It is used when the application recurrently deletes objects.
- *Heap 5*
  It is similar to the previous heap allocation scheme, but it also permits to combine non-contiguous memory blocks.

3.4.2 Dynamic and static allocation

FreeRTOS allows the application developer to allocate memory, creating objects like tasks, timers, queues, mutexes, semaphores without any dynamically allocated memory. Within a FreeRTOS application, both static allocation and dynamic allocation can be used, depending on the needs of the project. Below we describe pros and cons of the two memory allocation methods.
Dynamic allocation is simpler than static memory allocation and it often reduces RAM consumption. Using dynamic allocation inside of an API function, the memory is allocated automatically, the developer of the application does not deal with these details and the function call requires fewer parameters than static

allocation, making the code easier to read. When an object is eliminated, its RAM can be used again by another object and the memory allocation method can be selected depending on the application. A set of API functions that create FreeRTOS objects by means of dynamic memory allocation can be used only if the *configSUPPORT_DYNAMIC_ALLOCATION* is set to one (if undefined is set to one by default). Among them there are:

- xTaskCreate();
- xTimerCreate();
- xQueueCreate();
- xSemaphoreCreateMutex().

The application writer can have more control over memory management, using static allocation for assigning RAM to application objects. Static allocations allow to locate objects at specific memory areas, the maximum RAM usage is known at link time, while dynamic allocation works at run time, and this type of allocation lets to use the RTOS within a project that does not tolerate dynamic allocation. The application developer declares a proper object variable and passes the variable address to the API function. If the *configSUPPORT_STATIC_ALLOCATION* parameter, located in FreeRTOSConfig.h, is set to 1, these API functions are available to create objects by means of static memory allocation:

- xTaskCreateStatic();
- xTimerCreateStatic();
- xQueueCreateStatic();
- xSemaphoreCreateMutexStatic().

3.4.3 Time management

When a FreeRTOS task is blocked, it can specify a deadline beyond which it does not want to wait. If the task is in a sleep mode, it can indicate the time at which it must be awakened. The kernel of this real time operating system measures time by mean of a tick counter. An interrupt (the tick interrupt) increases the counter, assuring temporal precision. Time is measured by the kernel to a resolution depending on the indicated timer interrupt frequency. When the interrupt increments the tick count, the kernel verifies if there is a task that must be unblocked or awakened. In the case a woken task has a higher priority than the interrupted task (the task that was running), the tick Interrupt Service Routine returns to the task with the highest priority, executing a context switch and suspending the previous task. The ISR performs a pre-emption, that is a task is removed from the running state against its will.

At the end of each time slice the scheduler is executed and it chooses the task to run. The tick interrupt is a period interrupt that is raised for this reason. The time slice has not a fixed duration for all the applications, it can be set by changing the *config_TICK_RATE_HZ* parameter inside of *FreeRTOSConfig.h* file. Every FreeRTOS API function denotes time using ticks. If needed, a macro called *pdMS_TO_TICKS()* is used to convert a time from milliseconds to ticks. This function requires a unique parameter, the time expressed in milliseconds and returns the corresponding time in ticks. Time is handled by the operating system and the tick count indicates the amount of tick interrupts that have happened since the scheduler was started.

3.4.4 Delay

Within a task, polling is almost never a good solution, because the task uses processor time without performing any useful work. If a task must be suspended for a certain time, the optimal choice could be the use of the *vTaskDelay()* API function (below the function prototype).

```
void vTaskDelay(TickType_t xTicksToDelay);
```

This function is available only if in the config file, the *INCLUDE_vTaskDelay* is set to 1. After *vTaskDelay()* function call, the task enter in the blocked state for *xTicksToDelay* tick interrupts, the value passed as unique parameter. While the task is blocked, it does not require processor time, the processor is exploited by another task and, at the end of the delay period, the blocked task returns in the ready state.

As an alternative, the *vTaskDelayUntil()* API function requires two parameters that indicate the precise tick count value at which the blocked task (the task that calls the function) should return into the ready state. In this case the value of the tick time is an absolute time value, while the previous function parameter indicated a relative time, connected to the time in which the function was called. *vTaskDelayUntil()* function wants two parameters:

TickType_t* pxPreviousWakeTime	Since this function is used within a task that executes periodically, this parameter indicates at which time the task last exited from the blocked state. This variable must be initialized when it is used for the first time and subsequently is automatically updated.
TickType_t xTimeIncrement	It indicates the frequency with which the task is executed (in ticks)

Table 3.2: vTaskDelayUntil() API function parameters.

Below we describe a simple example, to better explain task and delay usage and to summarize what has been seen in this chapter. In the main() function two instances of the same task are created. Each instance will execute separately while the scheduler controls tasks execution. After the tasks creation, the scheduler is started. The task function, that implements an infinite loop, prints a string (passed to the task as parameter) and then delays for a certain period, before restarting its execution.

```
/* These strings will be passed to the task as parameters. */
static const char *task1Text = “Running task: Task 1\r\n”;
static const char *task2Text = "Running task: Task 2\r\n";

/* main function */
int main(void)
{
    /* Create the tasks. The first parameter is a pointer to the
    function that implements the task, the second is the task
    name, the other four parameters represent the stack depth,
    the value passed to the task function, the task priority and
    the handler to the task object. */
    xTaskCreate( taskFunction, "Task 1", 1000, (void*)task1Text,
    2, NULL );
    xTaskCreate( taskFunction, "Task 2", 1000, (void*)task2Text,
    5, NULL );
    /* Start the scheduler */
     vTaskStartScheduler();

    /* This point should be never reached.
*/
    return 0;
}

/* Task implementation function */
void taskFunction(void *inParameter)
{
    char *taskName;
    const TickType_t delay500ms = pdMS_TO_TICKS(500);

    /* The string to print is taken from the input parameter. */
    taskName = (char *) inParameter;

    /* This task is realized in an infinite loop. */
    for(;;)
    {
        /*  Print  the  name  of  the  task  (this  function  is  not
        implemented). */
        printString(taskName);

        /* Delay for 500ms. */
        vTaskDelay(delay500ms);
    }
}
```

Chapter 4

Hardware description

An embedded system is a computer system built specifically to perform a limited number of tasks. These systems are necessary components for the development of applications in the IoT world. Embedded systems usually are dedicated to special purpose applications, which often must comply with real-time constraints. A traditional computer can manage an almost unlimited amount of activities, while an embedded system typically has limited resources, and this fact prevents it to perform some tasks. But on the other hand, despite its small size, an embedded system guarantees high reliability, high performances, low energy consumptions and low costs.

The hardware of embedded systems is founded on the concept of microcontroller, which includes CPU's, memory and peripheral interfaces on a single chip. The constant evolution of microcontrollers complexity is due to the advancement of the integrated circuits technology. Nowadays, another optimal solution can be adopted, thanks to the introduction of System-On-Module (SoM), a complete embedded system integrated in a single board, ideal for low consumption applications. In a SoM multiple cores can coexist, allowing the implementation of Asymmetric Multi-Processing solutions.

In this chapter we introduce the hardware used for the testing phase, starting from VAR-SOM-MX7 board, a SoM which couples two distinct CPUs (a ARM Cortex-A7 equipped with Linux OS and a ARM Cortex-M4 core that runs FreeRTOS). We describe how to prepare the board in a context of AMP, and afterwards we introduce the MRF24J40 radio transceiver, a device that exploits the IEEE 802.15.4 stack, useful for a large field of IoT applications.

4.1 VAR-SOM-MX7 board

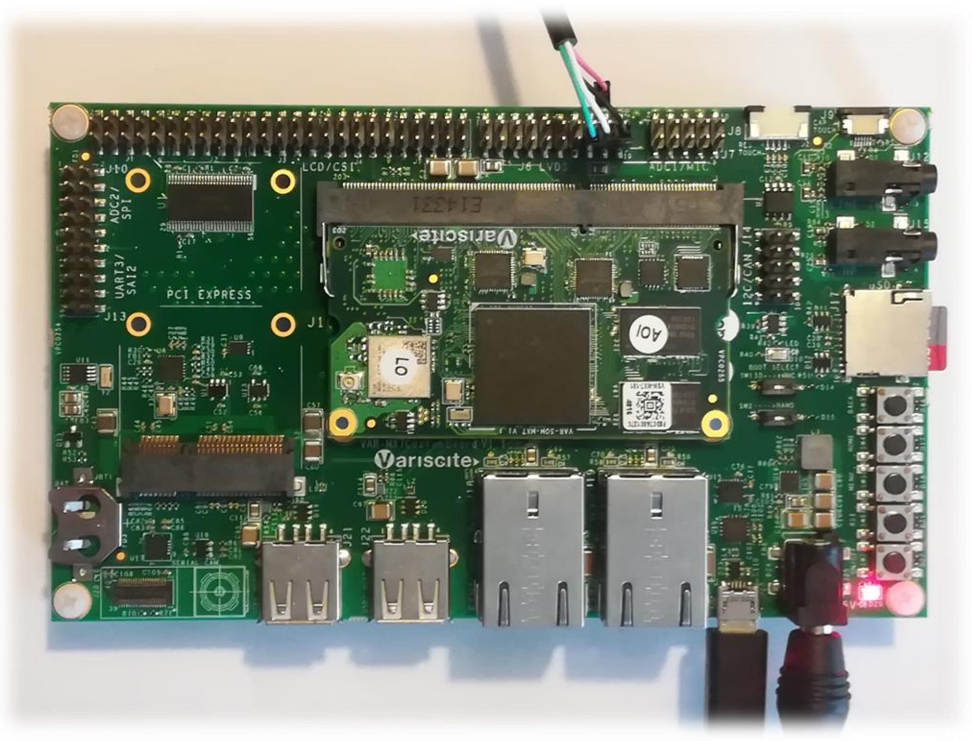


Figure 4.1: VAR-SOM-MX7 board.

The VAR-SOM-MX7 board is a System on Module based on the i.MX7-Dual processor, an ideal solution for IoT applications. The processor is composed by a dual core 1 GHz ARM Cortex-A7 and a real-time 200 MHz ARM Cortex-M4 co-processor. The co-processor improves the performance of the system, reducing the workload of the main processor. The board fulfils the demands of an embedded system, optimizing cost, size and power consumption. Besides it includes many connectivity options, like Bluetooth, Wi-Fi and Ethernet.

The heterogenous architecture of the processor allows to run a real-time operating system like FreeRTOS on the Cortex-M4 core, to manage real-time tasks, and a more complex OS (Linux) on the Cortex-A7 core. Cortex-A7 is an efficient processor used in a large set of devices, such as smartwatch, single board computers and wearables. Cortex-M4 core is used in applications that require high performances and low costs devices. It accelerates single precision floating-point operations and owns high signal processing capabilities.

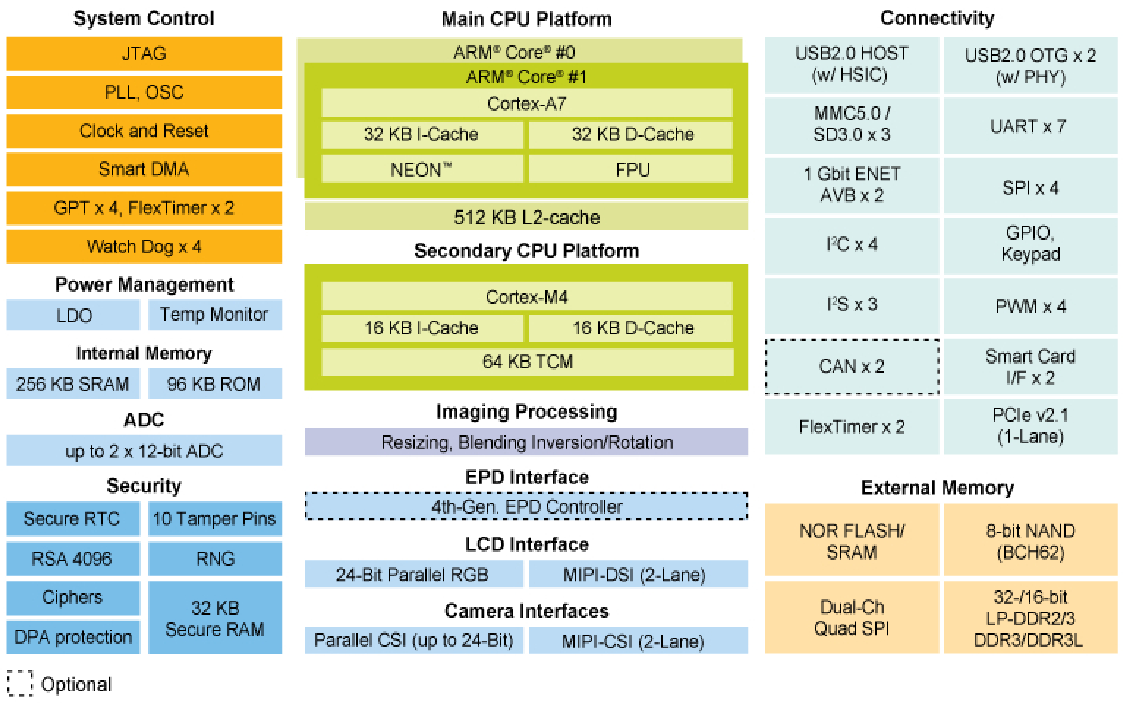


Figure 4.2: summary of VAR-SOM-MX7 characteristics.

4.2 Board preparation

4.2.1 Minicom

VAR-SOM-MX7 can be connected to a Linux computer through a serial USB port, and the communication with the board is possible by means of a serial communication program, like Minicom. An alternative for Windows users could be puTTY.

Minicom can be launched with the call:

```
$ sudo minicom -D <device_name>
```

In alternative, a previous setup is made, setting the device name, the baud rate and the hardware flow control. Now minicom is launched simply typing:

```
$ sudo minicom
```

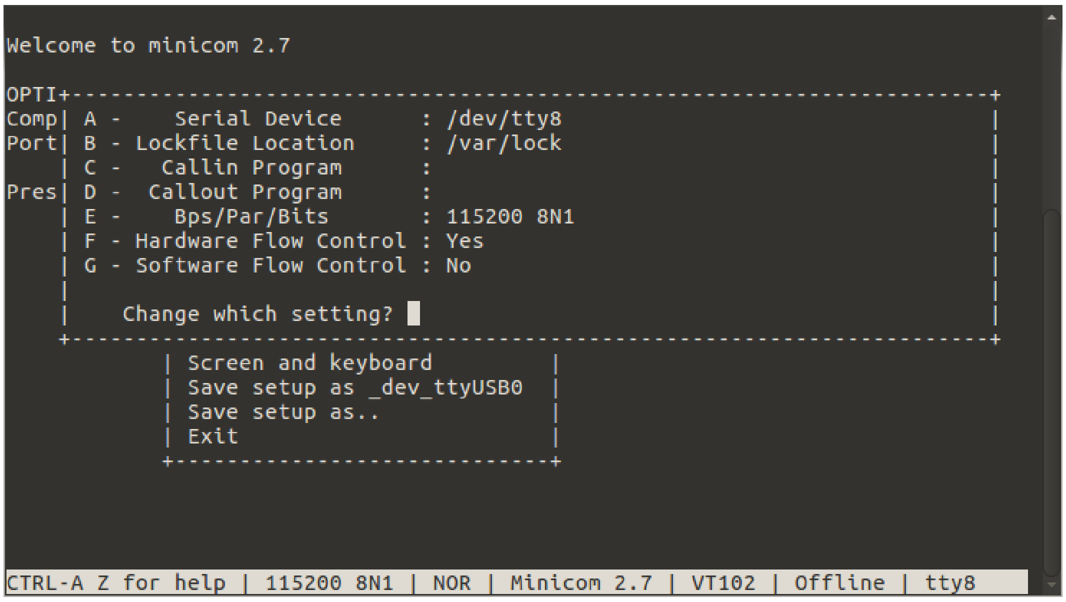
The program accepts a list of command line arguments:

`-h`	shows the list of arguments accepted by the program
`-b <baudrate>`	is the baud rate that the device uses

`-D <device name>` is the name of the serial device

To find the name of the tty port in Linux is sufficient to type the command:

```
$ dmesg | grep tty
```



```
Welcome to minicom 2.7

OPTI+-----------------------------------------------------------------------+
Comp| A -    Serial Device      : /dev/tty8                                 |
Port| B - Lockfile Location     : /var/lock                                 |
    | C -   Callin Program      :                                           |
Pres| D -  Callout Program      :                                           |
    | E -    Bps/Par/Bits       : 115200 8N1                                |
    | F - Hardware Flow Control : Yes                                       |
    | G - Software Flow Control : No                                        |
    |                                                                       |
    |    Change which setting?                                              |
    +-----------------------------------------------------------------------+
            | Screen and keyboard         |
            | Save setup as _dev_ttyUSB0  |
            | Save setup as..             |
            | Exit                        |
            +-----------------------------+




CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | tty8
```

Figure 4.3: minicom setup window.

4.2.2 Yocto

In the world of embedded devices, Linux is one of the most used operating systems, and it can be considered as a family composed by a great number of different releases. In this context there is the need of a unique tool that allows to make simple and standard the preparation of an OS build. Yocto is an open source project created to provide an optimal and customized Linux solution for every hardware architecture. It is born as a collaboration between electronics companies, operating systems developers and hardware vendors. The project provides a whole embedded development environment, consisting of tools and documentation, which permits to quickly create a personal distribution for a specific architecture. Yocto provides an updated and complete Linux OS, composed by a stable kernel, a reliable toolchain and consistent package versions. The support for a specific embedded architecture is distributed through BSP (Board Support Packages) layers, a collection of code expressly written for a hardware board.

We use Yocto to create a Linux distribution specific for the VAR-SOM-MX7 board. Since the host must be a Linux system, we use a machine equipped with the ubuntu 16.04 release. After installing a set of required packages in the host, download a yocto release (Yocto Pyro), setup the OpenEmbedded environment and run the bitbake command, which finally create the Linux image. OpenEmbedded environment is a build automation tool used for the creation of specific Linux distributions. If needed there is the opportunity to modify the *local.conf* file, that contains build configuration options. A more complete guide is provided by Variscite.

At this point, a specific VAR-SOM-MX7 Linux build is available. Inside tmp/deploy/images/imx7-var-som folder we can find the resulted images:

- *fsl-image-gui-imx7-var-som.sdcard*

  This is an image for the boot from SD card. It can be copied into an SD card, that is utilized to boot the system.
- *fsl-image-gui-imx7-var-som.tar.bz2*

  It is used to create a NFS (Network File System) root file system on the host, or to create an extended SD card.
- *fsl-image-gui-imx7-var-som.ubi*

  This is a UBI (Unsorted Block Image) image that contains a UBIFS volume. It is used for writing to the flash memory.
- *zImage*

  A Linux kernel image. This binary can be used indistinctly for SD card, eMMC and NAND flash.
- *u-boot.imx-sd*

  U-Boot built both for SD card and eMMC.
- *u-boot.imx-nand*

  U-Boot built for NAND memory.
- *zImage-imx7d-var-som-emmc.dtb*

  Device tree blob used for System on Modules with eMMC.
- *zImage-imx7d-var-som-nand.dtb*

  Device tree blob used for SOMs with NAND.

Now the user has two choices. He can flash the .sdcard image directly into a memory card or make a manual partition of the card and subsequently flash the image. The Sd card contains, at the beginning, a small memory unallocated space, saved for U-Boot. In the first partition, formatted with FAT16, there is a Linux image and the device tree blobs, while the second (ext4) partition contains the file system.

Once the SD card is ready, it can be used within the board. At first, the two boot dip switches of the board had to be setted in the correct way, following these instructions, to select boot mode:

Switch 1	**Switch 2**	
0	0	For the SD card boot
1	0	For the eMMC boot
0	1	For the NAND boot
1	1	Illegal combination

Table 4.1: Boot switches configuration.

For the SD card boot, switches must be setted with the 0 - 0 configuration. VAR-SOM-MX7 board is provided of NAND memory or eMMC, but not both. At this point the board is ready for the power up.

4.2.3 U-Boot

U-Boot (Universal Boot Loader) is an open source bootloader commonly used in embedded systems for supporting the boot of an operating system kernel, among which the Linux kernel. It is loaded from the SD card, it includes a menu for the interaction with the user and it takes care of all the steps for the OS boot in a large variety of devices.

The bootloader runs a command-line interface, which can be accessed through Minicom, or another serial communication program, thanks to the serial port of the board. The console is a fundamental component that allows the user to interact with U-Boot. The command prompt of the bootloader appears by pressing a key before the loading of the Linux kernel. This stops the booting process and allows the interaction with U-boot. The kernel can be loaded through this console and it is possible to do a lot of other operations: read flash memory, download files from a network (using DHCP, NFS, RARP, TFTP, BOOTP protocols) or from the serial port (by means of a serial communication program), manage a device tree. The device tree is a data structure that describes the hardware devices that are connected to the board. It is passed to the OS kernel at boot time. The configurations required by the user are managed through a set of environmental variables that are saved in the flash memory.

U-boot is very powerful and flexible. While computer bootloaders select the memory locations of the kernel in an automatic way, U-Boot needs to find the kernel and the device tree locations in memory, specifing the memory addresses as command line arguments. This is due to the fact that embedded systems don't support complex firmwares like BIOS or UEFI. The bootloader supports numerous filesystems, but it doesn't need to be able to read the filesystem that it passes to the kernel, it transfers the filesystem as a simple parameter without understanding the content of the data.

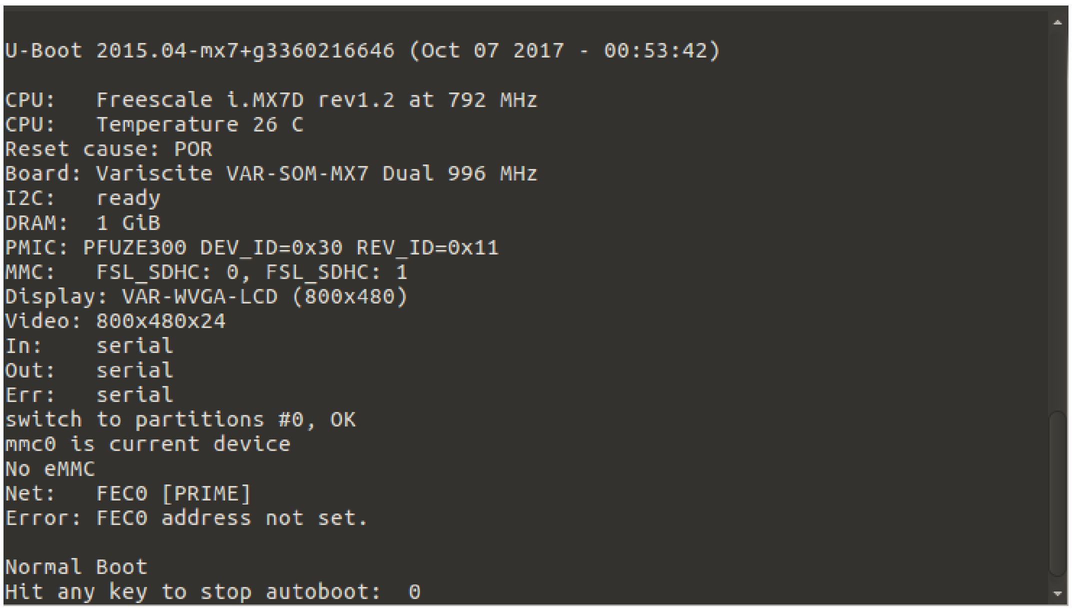

```
U-Boot 2015.04-mx7+g3360216646 (Oct 07 2017 - 00:53:42)

CPU:   Freescale i.MX7D rev1.2 at 792 MHz
CPU:   Temperature 26 C
Reset cause: POR
Board: Variscite VAR-SOM-MX7 Dual 996 MHz
I2C:   ready
DRAM:  1 GiB
PMIC: PFUZE300 DEV_ID=0x30 REV_ID=0x11
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: VAR-WVGA-LCD (800x480)
Video: 800x480x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
No eMMC
Net:   FEC0 [PRIME]
Error: FEC0 address not set.

Normal Boot
Hit any key to stop autoboot:  0
```

Figure 4.4: U-Boot command line interface.

4.2.4 Build and run an application

After the installation of GNU-ARM bare-metal toolchain and of FreeRTOS BSP on the computer host, a lot of FreeRTOS examples are available to test board functionalities. FreeRTOS Board Support Packages includes peripheral drivers, an open source event driven preemptive real time operating system and a multicore communication stack (called RPMsg). Below we will describe how to build and run a simple FreeRTOS application.

When VAR-SOM-MX7 is turned on, the Cortex-A7 always boots as primary core, since the other core, the Cortex-M4, doesn't possess a boot ROM. Cortex-A7 core has to load the firmware of the Cortex-M4 and to launch it. Cortex-A7 also enables M4 clock and clears its reset bit.

FreeRTOS BSP allow the Cortex-M4 core to use three different types of memory areas, as described in this table:

Memory type	**Cortex-M4 memory area**	**Cortex-A7 memory area**	**Memory length**
DDR	0x9FF00000-0x9FF07FFF	0x9FF00000-0x9FF07FFF	32 kB
OCRAM (part of OCRAM_128KB)	0x20210000-0x20217FFF	0x00910000-0x00917FFF	32 kB
TCM (TCML)	0x1FFF8000-0x1FFFFFFF	0x007F8000-0x007FFFFF	32 kB
data section (TCMU)	0x20000000-0x20007FFF	0x00800000-0x00807FFF	32 kB

Table 4.2: Cortex-M4 memory areas.

Build an application

In FreeRTOS, to build a simple application, the following steps must be followed:

- enter in the application folder
- enter in the armgcc folder with the command
  ```
  $ cd armgcc
  ```
- execute these commands

```
$  export  ARMGCC_DIR=~/var-som-mx7_m4-freertos/gcc-arm-
none-eabi-5_4-2016q3
$ ./build_all.sh > /dev/null
```

Finally, the .bin file, generated during the build phase, must be copied inside the FAT partition of the boot sd card.

Run an application

If we rely on the default boot process, when we power up the VAR-SOM-MX7 board, U-Boot automatically sets a device tree blob, starting Cortex-A7 core, without using Cortex-M4. U-Boot offers the possibility to load a different device tree blob (dtb), to allow both Cortex-A7 and Cortex-M4 cores accessing shared resources, like RAM, SPI and I2C. To perform it, these steps must be followed:

- before turning on the board, connect the host computer with the board, through two serial USB ports, then open two serial terminals (minicom). They are used to communicate with the two cores.
- check if the boot select switches are sets on the boot from SD mode (0 - 0: for the SD card boot)
- insert the SD card in the VAR-SOM-MX7 SD slot
- turn on the board
- when U-Boot starts, press any key to stop the boot process
- enable Cortex M4 core:
  ```
  $ setenv use_m4 yes
  ```
- initialize m4bootdata variable with the adress of the memory (TCM memory case)
  ```
  $ setenv m4bootdata 0x007F8000
  ```
- initilize m4image variable with the application name (rpmsg_str_echo_freertos_example case)
  ```
  $ setenv m4image rpmsg_str_echo_freertos_example.bin
  ```
- save U-Boot environment
  ```
  $ saveenv
  ```
- start Cortex-M4run
  ```
  $ m4boot
  ```

- start Cortex-A7

  ```
  $ run bootcmd
  ```

4.2.5 StrEcho FreeRTOS application

StrEcho application demonstrates how the RPMsg remote peer stack works on FreeRTOS. At first, the communication channel must be created. Afterwards, the master (Cortex A7, equipped with Linux) waits for a user input string. When the user inserts a string, the RPMsg virtual tty sends it to the Cortex-M4 (which runs FreeRTOS). M4 core shows in the minicom terminal the received string and its length, then the same message is sent back to the Cortex-A7 core, as a kind of acknowledgement. A7 core displays the string and waits until user sends a new message. The loop continues to demonstrate RPMsg's ability to send arbitrary content.

When the cortex-M4 boot process succeeds, these strings are displayed in the terminal:

```
RPMSG String Echo FreeRTOS RTOS API Demo...
RPMSG Init as Remote
```

After Linux boots, the RPMsg tty module must be installed, typing:

```
$ modprobe imx_rpmsg_tty
```

Cortex-M4 terminal confirms the correctness of the installation.

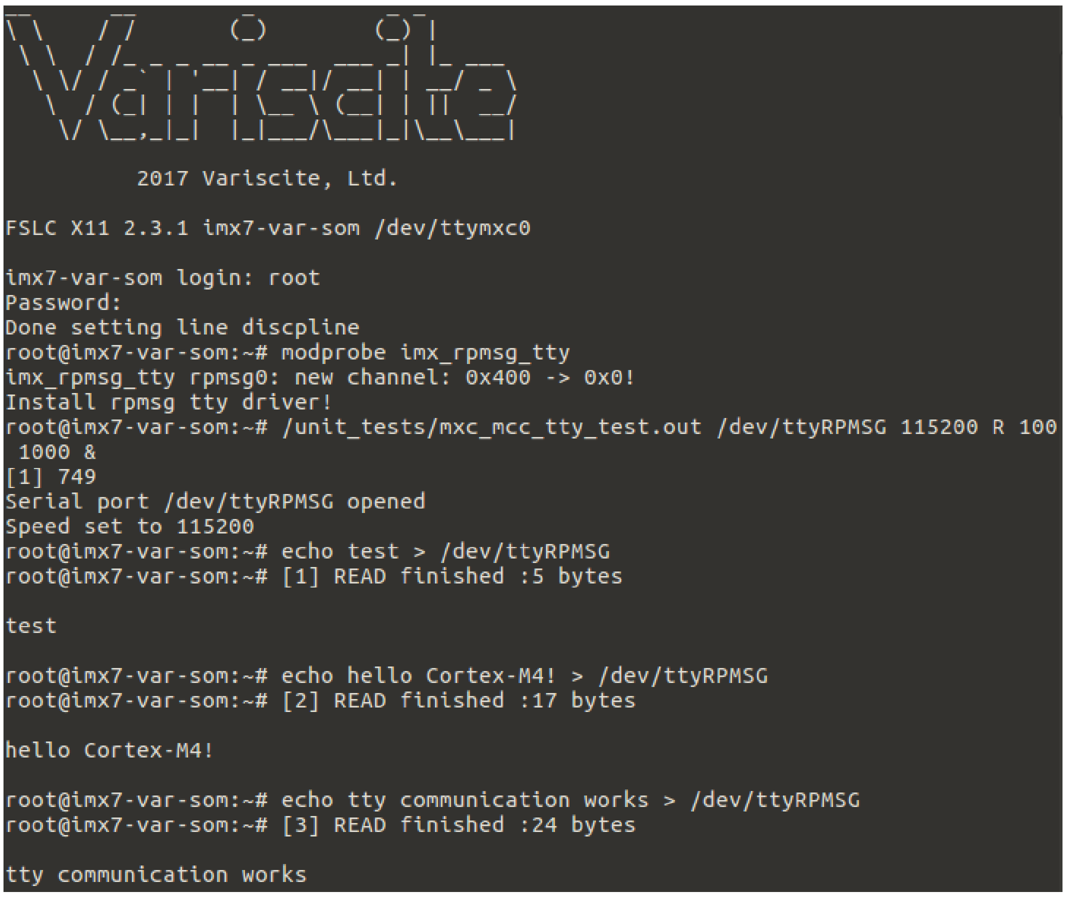
```
Name service handshake is done, M4 has setup a rpmsg
channel [1 ---> 1024]
```

The Cortex-M4 core channel address is 1, the Cortex-A7 channel address is 102

Then the RPMsg tty receive program can be launched:

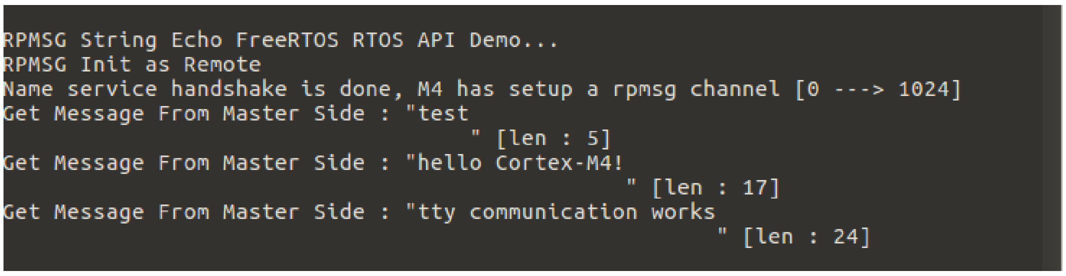
```
$ /unit_tests/mxc_mcc_tty_test.out /dev/ttyRPMSG 115200 R 100
1000 &
```

Only at this point the user can insert a string in the tty, using the echo command, as shown in Figure 4.5.



```
__     __         _          _ _
\ \   / /        (_)        (_) |
 \ \ / /_ _ _ __ _ ___  ___ _| |_ ___
  \ V / _` | '__| / __|/ __| | __/ _ \
   \ / (_| | |  | \__ \ (__| | ||  __/
    V \__,_|_|  |_|___/\___|_|\__\___|

          2017 Variscite, Ltd.

FSLC X11 2.3.1 imx7-var-som /dev/ttymxc0

imx7-var-som login: root
Password:
Done setting line discpline
root@imx7-var-som:~# modprobe imx_rpmsg_tty
imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x0!
Install rpmsg tty driver!
root@imx7-var-som:~# /unit_tests/mxc_mcc_tty_test.out /dev/ttyRPMSG 115200 R 100
 1000 &
[1] 749
Serial port /dev/ttyRPMSG opened
Speed set to 115200
root@imx7-var-som:~# echo test > /dev/ttyRPMSG
root@imx7-var-som:~# [1] READ finished :5 bytes

test

root@imx7-var-som:~# echo hello Cortex-M4! > /dev/ttyRPMSG
root@imx7-var-som:~# [2] READ finished :17 bytes

hello Cortex-M4!

root@imx7-var-som:~# echo tty communication works > /dev/ttyRPMSG
root@imx7-var-som:~# [3] READ finished :24 bytes

tty communication works
```

Figure 4.5: StrEcho FreeRTOS application on Cortex-A7.



```
RPMSG String Echo FreeRTOS RTOS API Demo...
RPMSG Init as Remote
Name service handshake is done, M4 has setup a rpmsg channel [0 ---> 1024]
Get Message From Master Side : "test
                                    " [len : 5]
Get Message From Master Side : "hello Cortex-M4!
                                                " [len : 17]
Get Message From Master Side : "tty communication works
                                                       " [len : 24]
```

Figure 4.6: StrEcho FreeRTOS application on Cortex-M4.

4.3 MRF24J40 radio transceiver

The transceiver is a device able to transmit and receive signals. It includes a transmitter and a receiver, which are enclosed within a single electrical circuit. Transceivers can be divided in two categories, namely full duplex and half duplex transceivers. A half duplex transceiver does not allow to receive a signal during transmission, hence while the transceiver transmits a message, the receiver is disabled. Conversely a full duplex transceiver can receive and send signals at the same time.

The MRF24J40 chip is a full duplex wireless radio transceiver that works with a 2.4 GHz frequency. This wireless device is suitable for many embedded systems, since its hardware structure allows to reduce production costs and energy consumption. This low data rate WPAN device is structured to respect the IEEE 802.15.4 standard, used by ZigBee and MiWi protocols. The standard integrates the MAC and PHY layers, that are the starting point for a wireless network device. The Media Access Controller circuitry can verify if IEEE 802.15.4 packets are received and if the format of the packets is correct, with data that is subsequently buffered in transmit and receive FIFOs.

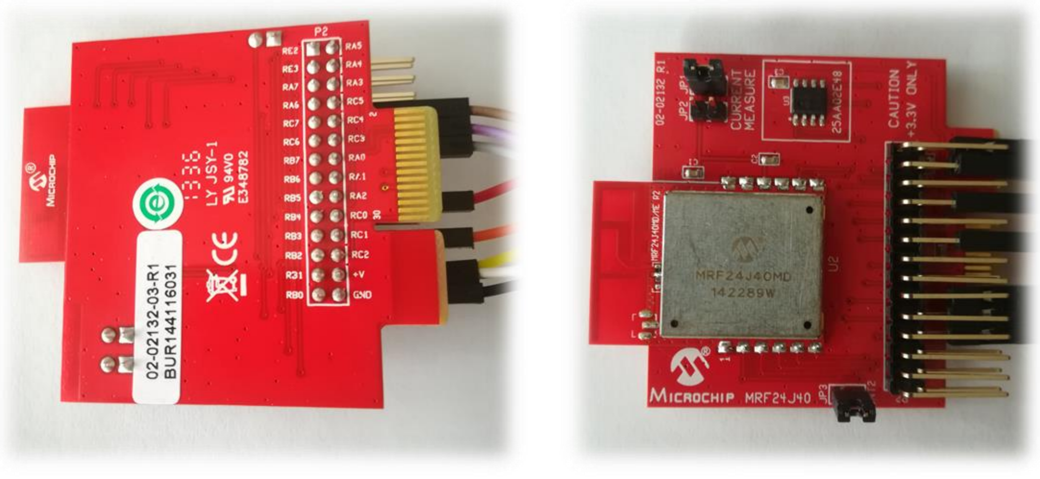


Figure 4.7: MRF24J40 radio transceiver.

One of the most important parameters of a radio transceiver is the value of sensitivity, a feature that indicates the maximum noise beyond which the device doesn’t receive the correct signals. The MRF24J40 is one of the best solutions on the market, with a -95 dBm sensitivity. The transceiver also includes a hardware component, the Received Strength Signal Interface (RSSI), which can supply a value that shows the intensity of the received signal. Considering that the transceiver can modulate output power, the RF channel can be optimized, sending only the necessary power to avoid useless noise. The device allows to minimize power consuption and it provides a low current sleep mode (2 μA).

MRF24J40 works with a 20 MHz oscillator, and it includes also a PLL, hooked to the oscillator, to improve the accuracy of data synchronization. MRF24J40 memory can be divided in many types, among which short address control registers (used for the component settings), long address control registers (used for RF and clock settings), transmission and reception buffers and the security buffer.

The transceiver consists of four functional blocks:

- A serial 4-wire SPI interface necessary for communication between the device and the host controller (the VAR-SOM-MX7 board).
- Control registers available for the user of the host controller, which can set MRF24J40 parameters.
- The MAC module, that implements the IEEE 802.15.4 standard.
- The physical driver.

There are also six GPIO pins that the user can set with the purpose of controlling the device.

4.3.1 Technical features

These are the main features of the MRF24J40 transceiver, extracted from the official datasheet.

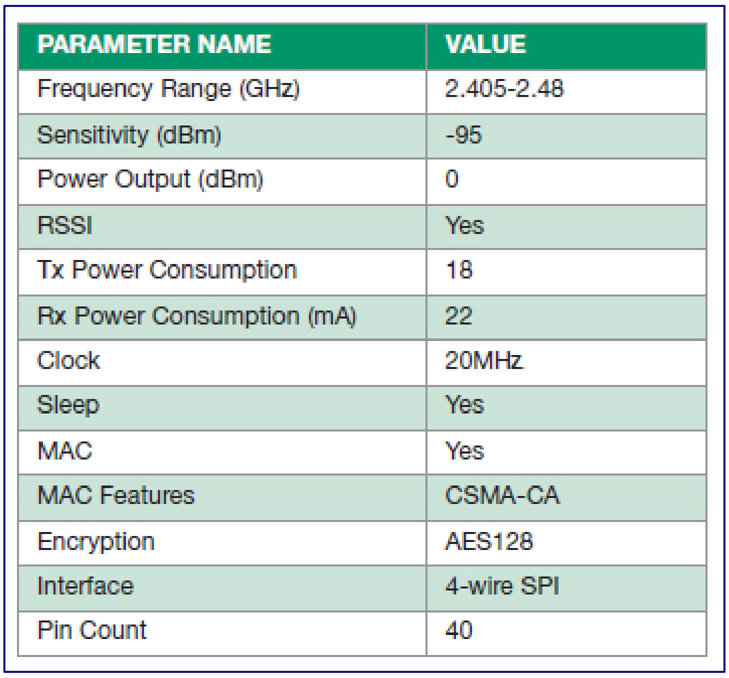
RF/Analog Features:

- ISM Band 2.405-2.48 GHz Operation
- Data Rate: 250 kbps (IEEE 802.15.4); 625 kbps (Turbo mode)
- -95 dBm Typical Sensitivity with +5 dBm Maximum Input Level
- +0 dBm Typical Output Power with 36 dB TX Power Control Range
- Differential RF Input/Output with Integrated TX/RX Switch

- Integrated Low Phase Noise VCO, Frequency Synthesizer and PLL Loop Filter
- Digital VCO and Filter Calibration
- Integrated RSSI ADC and I/Q DACs
- Integrated LDO
- High Receiver and RSSI Dynamic Range

MAC/Baseband Features:

- Hardware CSMA-CA Mechanism, Automatic Acknowledgement Response and FCS Check
- Independent Beacon, Transmit and GTS FIFO
- Supports all CCA modes and RSSI/ED
- Automatic Packet Retransmit Capability
- Hardware Security Engine (AES-128) with CTR, CCM and CBC-MAC modes
- Supports Encryption and Decryption for MAC Sublayer and Upper Layer



PARAMETER NAME	VALUE
Frequency Range (GHz)	2.405-2.48
Sensitivity (dBm)	-95
Power Output (dBm)	0
RSSI	Yes
Tx Power Consumption	18
Rx Power Consumption (mA)	22
Clock	20MHz
Sleep	Yes
MAC	Yes
MAC Features	CSMA-CA
Encryption	AES128
Interface	4-wire SPI
Pin Count	40

Figure 4.8: general MRF24J40 features

4.3.2 Features explanation

Below there will be a brief explanation of some technical concepts, listed in the device features, related to radio communication.

- *Frequency range:* RF (Radio Frequency) is a parameter that indicates the rate of an electrical oscillation. To receive a radio signal an antenna is required, coupled with a radio tuner (typically a resonator). The resonator amplifies frequencies in the desired range, the frequency range, and reduces the oscillations of other frequencies. The frequency range of MRF24J40 varies around 2.4 Ghz.
- *Sensitivity:* in a radio transceiver device, sensitivity indicates the lower power level of an input signal needed to produce an output signal which has an acceptable signal-to-noise ratio. Sensitivity is a receiver characteristic, it does not depend on the transmitter. If a receiver has a good sensitivity value, the transmission range growes and the receiver detects more distant and weaker signals.
- *Power output:* it is a specific characteristic of the transmitter that indicates the amount of power energy produced at the output.
- *RSSI:* Received Signal Strength Indicator is a parameter that measures the power of a received radio signal. The strength of a signal can condition the performance of a wireless network.
- *Phase Noise:* in signal processing phase noise happens if the phase of signals produced within a system is disturbed, making difficult to separate the desired frequency from unwanted signals close to it.
- *PLL:* Phase-Locked Loop is a control system that produces an output signal which has a phase connected to that of the input signal. This fact implies that the input and output frequencies must also be identical. A PLL is integrated into many electronic devices, and it used for a lot of purposes, such as recovery a signal from a noisy channel, demodulate a signal, spread timed clock pulses in a microprocessor.
- *AES 128 encryption:* AES (Advanced Encryption Standard) is an encryption mechanism used for electronic data. It was first adopted by the government of USA (to overcome the limits of the previous Data Encryption Standard),

but thanks to its robustness and effectiveness, nowadays it is utilized all over the world. This is a symmetric-key algorithm, namely the same key allows both to encrypt and to decrypt. 128 indicates the key length.

Chapter 5

Software development

Within this thesis we have previously seen how the IEEE 802.15.4 protocol is the most suitable for the realization of an IoT network that requires low complexity, low power consumption, low data-rates, short communication ranges and reduced costs. We have subsequently described FreeRTOS, that is one of the most used Real-Time Operating Systems, especially within embedded systems. Its success is due to the simplicity of the kernel code, in addition to the fact that it is supported by a lot of hardware architectures. After preparing the hardware environment, the last step necessary to include the protocol stack within a FreeRTOS application is the porting phase. Porting indicates the process of transposing a software component to allow it to be used into a different environment.

The FreeRTOS source code is very simple and every additional feature must be attached to the source code through an add-on module. If we desire to use a protocol stack within an application, we must add the stack module to FreeRTOS. In this chapter we describe the software development steps. We firstly explain how to adapt the MRF24J40 device driver to the operating system. The driver provides a communication interface between the VAR-SOM-MX7 board and the radio transceiver. After that we examine the details of the IEEE 802.15.4 protocol software.

5.1 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a synchronous four wire communication system, used to connect a microcontroller with another device (a sensor, a peripheral, a microcontroller). It is a standard communication bus, widely used also in embedded systems. Transmission occurs between a master and one or more slaves. The master manages the bus, it emits the clock signal and it decide when starting and ending the communication. It is a serial bus, it is synchronous because a clock coordinates bit transmission and reception, and it establishes the transmission speed. It is either full-duplex since transmission and reception may occur at the same time. The transmission speed, that derives from the clock frequency, does not have a lower limit, while the maximum transmission speed value must be searched in the device datasheet. The SPI bus is a four-wire system, considering that the bus consists of four logical signals (excluding the reference connection, GND). These four signals are:

- SCLK (SCK) Serial Clock. It is an output from the master.
- MISO Master Input Slave Output.
- MOSI Master Output Slave Input.
- nCS (SS) Chip Select. It is an output from the master, used to decide with which slave device it wants to communicate.

The SCLK signal therefore represents the serial clock which decides the instant in which bits are output and read on the serial lines. This signal is issued by the master, consequently the master chooses to request the transmission of a word. Through MISO the device receives the serial data and at the same time, the device issues its output on the MOSI line.

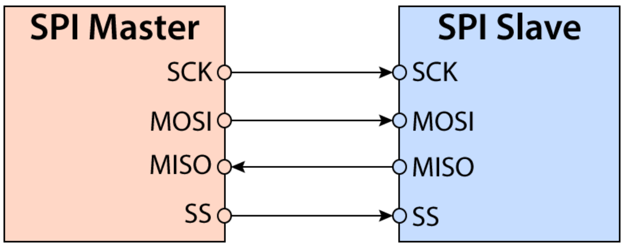


Figure 5.1: SPI bus scheme.

General purpose input-output (GPIO) is a pin on an integrated circuit that can be controlled by the user at run time. GPIO pins are not used by default and they do not have a specific purpose. GPIO pins are added to a chip when some additional control lines are needed, avoiding the problem of adding a set of other components to the circuit. GPIO pins can be configured to be both input or output pins and each pin can be enabled or disabled. The input values are readable, the value of a pin usually can be high or low, and these values can be utilized as IRQs. The output values can instead be both readable and writable. GPIOs are used within the devices that include a limited number of pins. Embedded applications benefit from GPIO pins, since these pins are used for reading from a lot of environmental sensors and for writing output to other peripherals.

The VAR-SOM-MX7 board uses the four wire SPI bus to communicate with the mrf24j40 radio transceiver. In addition to this, the transceiver interfaces with the board through two other pins: reset and interrupt. Two GPIO pins of the VAR-SOM-MX7 are configured for this purpose (J13.02 and J13.04 pins), as described in the following table.

Function	MRF24J40 connector	VAR-SOM-MX7 connector	Wire color
MOSI	P2.08	J10.07	Brown
MISO	P2.10	J10.05	Purple
SCLK	P2.12	J10.03	Gray
nRST	P2.17	J13.02	Red
nCS	P2.21	J10.09	Orange
INT	P2.25	J13.04	Yellow
3.3 V	P2.26	J10.01	White
GND	P2.28	J10.10	Black

Table 5.1: Pin settings for the connection between the transceiver and the board.

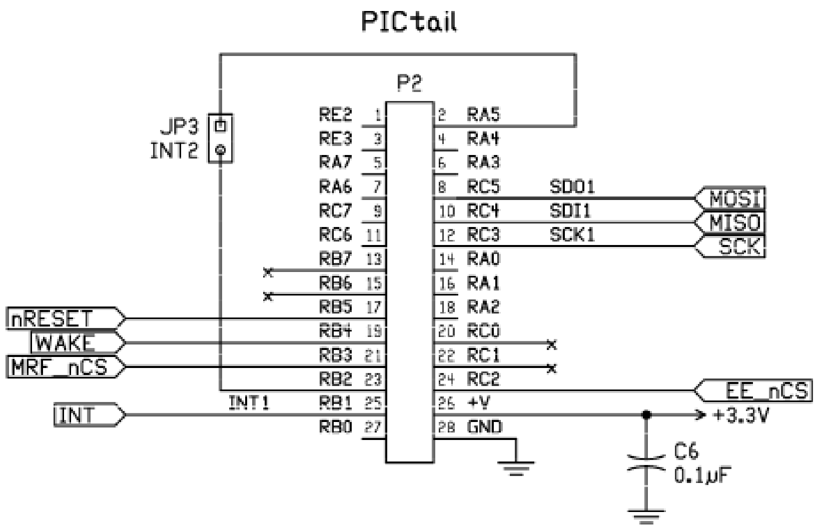


Figure 5.2: P2 pins scheme of the MRF24J40 radio transceiver.

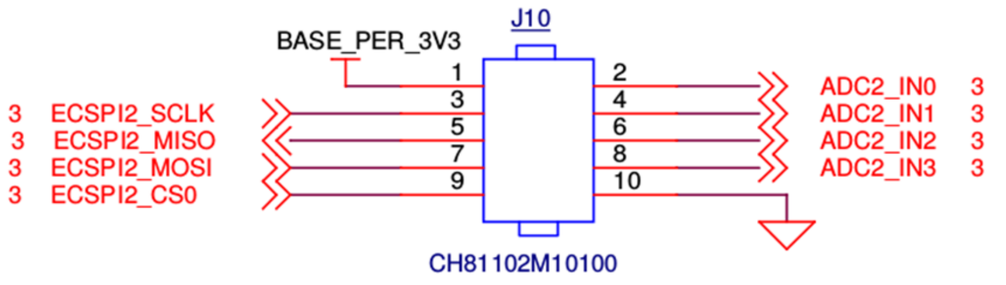


Figure 5.3: J10 pins scheme of the VAR-SOM-MX7 board.

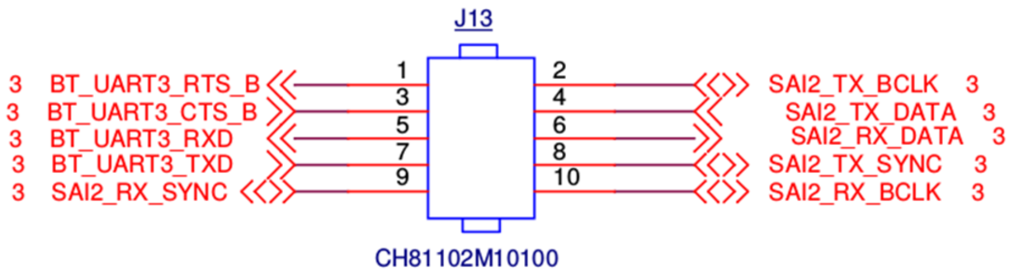


Figure 5.4: J13 pins scheme of the VAR-SOM-MX7 board.

5.2 Timers

A FreeRTOS application that includes the IEEE 802.15.4 protocol stack needs of a system for handling the timing. The main functions for which timers will be utilized are setting up delays and performing specific actions at certain times. Into an application we can use more than one timer, each of which will be created for a particular purpose. Timers are used both within the driver of the transceiver and within the stack. Below we describe the code that we have implemented to manage timers.

```
/* Enumerate available timers. */
enum _hw_timer {
 HW_TIMER_A = 0,
 HW_TIMER_B = 1,
 HW_TIMER_MAX = 2
};
```

```
/* i.MX timer initialize structure. */
typedef struct _timer_config
{
GPT_Type* base;
IRQn_Type IRQn;
SemaphoreHandle_t xSemaphore;
void (*callback)(void);
} timer_config_t;

/* Declaration of two timers, that will be used within the stack */
static timer_config_t timer_list[] = {
{
   .base = BOARD_GPTA_BASEADDR,
   .IRQn = BOARD_GPTA_IRQ_NUM
},
{
   .base = BOARD_GPTB_BASEADDR,
   .IRQn = BOARD_GPTB_IRQ_NUM
},
};

/* Initialize hardware timer, must be called before
Hw_Timer_Action() */
int Hw_Timer_Init(uint32_t timer, void (*func)(void))
{
    gpt_init_config_t config = {
        .freeRun    = false,
        .waitEnable = true,
        .stopEnable = true,
        .dozeEnable = true,
        .dbgEnable  = false,
        .enableMode = true
    };

  if (timer >= HW_TIMER_MAX)
   return -1;

    /* Initialize GPT module */
    GPT_Init(timer_list[timer].base, &config);

    /* Set GPT clock source to 24M OSC */
    GPT_SetClockSource(timer_list[timer].base, gptClockSourceOsc);

    /* Set GPT interrupt priority 3 */
    NVIC_SetPriority(timer_list[timer].IRQn, 3);

    /* Enable NVIC interrupt */
    NVIC_EnableIRQ(timer_list[timer].IRQn);

   timer_list[timer].callback = func;

    if (timer_list[timer].callback == NULL)
        timer_list[timer].xSemaphore = xSemaphoreCreateBinary();
```

```
return 0;
}

/* If no callback has been provided at init, block task for some
time with hardware timer, otherwise exit immediately and the
callback will be called when the timer expires. This timer is not
multi-thread safe and could only be called in one task.
 @param us microseconds to delay */
int Hw_Timer_Action(uint32_t timer, uint32_t us)
{
    /* First get the counter needed by delay time */
    uint64_t counter = 24ULL * us;
    uint32_t high;
    uint32_t div24m, div;

   if (timer >= HW_TIMER_MAX)
   return -1;

   if (us == 0)
   return 0;

    /* Get the value that exceed maximum register counter */
    high = (uint32_t)(counter >> 32);

    /* high could not exceed 24000, so that predivider is enough */
    /* We need PRESCALER24M only if high exceed PRESCALER maximum
       value */
    div24m = high / 4096;
    /* Get PRESCALER value */
    div = high / (div24m + 1);

    /* Now set prescaler */
    GPT_SetOscPrescaler(timer_list[timer].base, div24m);
    GPT_SetPrescaler(timer_list[timer].base, div);

    /* Set GPT compare value */
    GPT_SetOutputCompareValue(timer_list[timer].base,
              gptOutputCompareChannel1,
              (uint32_t)(counter / (div24m + 1) / (div + 1)));

    /* Enable GPT Output Compare1 interrupt */
    GPT_SetIntCmd(timer_list[timer].base,
              gptStatusFlagOutputCompare1, true);

    /* GPT start */
    GPT_Enable(timer_list[timer].base);

    if (timer_list[timer].callback == NULL) {
        /* Wait until GPT event happens. */
        xSemaphoreTake(timer_list[timer].xSemaphore,
                           portMAX_DELAY);
    }
    return 0;
}

/* Handler for HW_TIMER_A. An equivalent function exists for the
management of HW_TIMER_B */
void BOARD_GPTA_HANDLER(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    uint32_t timer = HW_TIMER_A;

    /* When GPT time-out, we disable GPT to make sure this is a
    one-shot event. */
    GPT_Disable(timer_list[timer].base);
    GPT_SetIntCmd(timer_list[timer].base,
                gptStatusFlagOutputCompare1, false);
    GPT_ClearStatusFlag(timer_list[timer].base,
                gptStatusFlagOutputCompare1);

    if (timer_list[timer].callback == NULL) {
        /* Unlock the task to process the event. */
        xSemaphoreGiveFromISR(timer_list[timer].xSemaphore,
                    &xHigherPriorityTaskWoken);
    } else {
        timer_list[timer].callback();
    }

    /* Perform a context switch to wake the higher priority task.
    */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

5.3 Porting of the MRF24J40 driver

A device driver is a program that manages a specific device that is connected to a computer or a microcontroller. The main purpose of a driver is to provide a software interface to the hardware device. In this way, the operating system and any other program can use hardware functionalities through the software interface, without knowing every detail of the hardware device. The driver communicates with the device by means of a bus, that connects the hardware to the computer. When a computer program calls a driver function, the driver forwards the request to the hardware device. If a device sends data to the driver, the driver can call a function located within the initial program. Every hardware device needs of a driver and the driver is a software component dependent on the operating system. Thanks to device drivers, the application developer can write a program that works with every specific hardware. It is sufficient to change the driver, while the application code is hardware-independent.

The first step to develop a FreeRTOS application that uses the functionalities of the MRF24J40 radio transceiver, is the porting of the device driver within a FreeRTOS environment, through the addition of an independent module. In this section we describe the source files, reporting the basic functions used for the device configuration and for the communication with the transceiver. The device driver consists of the following source files:

- mrf24j40_compiler.h
- mrf24j40.h
- mrf24j40.c
- mrf24j40_hal.h
- mrf24j40_hal.c

Below we describe the details of each file.

5.3.1 Files description

mrf24j40_compiler.h

This file simply includes stdint (for intX_t and uintX_t data types) and stdlib (for the NULL data type).

mrf24j40.h

The mrf24j40 transceiver contains two distinct memory areas, named short address memory and long address memory. The first area includes only control registers, the second area, which is bigger, contains also FIFO buffers. Control registers describes the operating parameters and the state of the transceiver, while FIFO are used as temporary transmission and reception buffers. When we need to access a memory area through the SPI port, we must transmit the address and the data that must be transmitted. For accessing to a short address memory area, the first byte transmitted through the SPI bus is composed, starting from the most significant bit, by a bit ‘0’, a 6 bits address and a ‘0’ or a ‘1’, respectively for a reading or a writing operation. The second byte indicates the value that must be written in the desired location. For accessing to the long address memory, which requires 12 bits addresses, a communication consists of 3 bytes. The most significant bit will be a ‘1’, followed by the 12 bits address, a ‘0’ or a ‘1’ to define the operation type and 2 bits that will be ignored. The third byte holds the value that must be written in the register. In both cases the last byte is ignored if the access to the register is due to a reading operation.

This file declares the basic functions of the driver. It does not depend on the operating system, it manages addresses and all the transmission parameters. The file firstly defines long and short address registers, channel setting codes, register setting masks and transmission power levels masks. After that, within this file a set of inline functions are defined. In C language an inline function, preceded by the keyword *inline*, is a function that provides a directive to the compiler. The function body is replaced inline by the compiler, namely the code of the function is inserted at the address of each function call. This operation removes the overhead generated by the function call.

The first basic inline functions included into this file are used to write a value, passed to the function through *val* parameter, respectively in the short and long address register *addr* (we show the function prototypes):

```
inline void mrf24j40_set_short_add_mem(uint8_t addr, uint8_t val);

inline void mrf24j40_set_long_add_mem(uint16_t addr, uint8_t val);
```

The content of the register *addr* can be also read:

```
inline uint8_t mrf24j40_get_short_add_mem(uint8_t addr);

inline uint8_t mrf24j40_get_long_add_mem(uint16_t addr);
```

These functions, within their body, call other hardware specific functions, *mrf24j40_hal_spi_read()* and *mrf24j40_hal_spi_write()*, used for reading and writing operations through the SPI bus. They are defined in the mrf24j40_hal.c file, which will be described later. The functions that we have described before, are used to initialize and manage network parameters, by reading and writing from registers. In the following examples, which define other inline functions present in the file mrf24j40.h, their use will be better understood.

```
/* This call sets the transmission power of the transceiver. */
inline void mrf24j40_set_pa(uint8_t pwr) {
	mrf24j40_set_long_add_mem(MRF24J40_RFCON3, pwr);
}

/* This function returns the transmission power of the radio
transceiver. */
inline uint8_t mrf24j40_get_pa() {
	return mrf24j40_get_long_add_mem(MRF24J40_RFCON3);
}

/* This routine starts the transmission of a message by the
transceiver. It transmits the packet from the normal transmission
FIFO, without security and acknowledgement request. */
inline void mrf24j40_set_tx() {
	mrf24j40_set_short_add_mem(MRF24J40_TXNCON, 0x01);
}

/* This function sets the MAC PAN ID in the transceiver. */
inline void mrf24j40_set_PAN_id(uint16_t id) {
	mrf24j40_set_short_add_mem(MRF24J40_PANIDL, (uint8_t)id);
	mrf24j40_set_short_add_mem(MRF24J40_PANIDH,
	                                        (uint8_t)(id>>8));
}
```

In addition to these, other routines take care of flushing the receive FIFO, set the communication channel, set the short and the extended MAC addresses in the radio transceiver, get the status of the radio, get the RSSI (received signal strength indication) value, set CSMA parameters.

mrf24j40.c

This source file defines a group of routines, among which a first function that initializes the radio transceiver:

```
int8_t mrf24j40_init(uint8_t int_setup, uint8_t ch, uint8_t port);
```

It returns 0 if the initialization goes well, otherwise it returns -1. Some other functions are useful for message management:

```
int8_t mrf24j40_store_norm_txfifo(uint8_t* buf, uint8_t len);
uint8_t mrf24j40_get_norm_txfifo(uint8_t pos);
uint8_t mrf24j40_get_fifo_msg(uint8_t *msg);
```

The first routine stores a buffer of *len* bytes in the transmission FIFO buffer of the transceiver, the second retrieves a byte at position *pos* of the same buffer, the third recovers a message from the reception FIFO buffer. Moreover, this driver file makes available routines for putting the radio into sleep mode, for awakening the radio and for setting the callbacks that must be executed at the end of a message transmission and when a message is received.

mrf24j40_hal.h

The Hardware Abstraction Layer (HAL) is a set of functions which allows a program to access directly to the hardware resources. It permits to hide the differences between different hardware devices. If a program is equipped with a Hardware Abstraction Layer, its portability on different devices and operating systems increases, and this is fundamental for embedded systems. Any changes in the program are exclusively done within the HAL, while the other code often remains the same in every developing environment.

This file contains only the declarations of the functions that are defined within mrf24j40_hal.c.

mrf24j40_hal.c

This file includes a group of hardware-specific functions. This code is specifically written for the VAR-SOM-MX7 board and it allows to perform the following tasks:

- Initialize the Hardware Abstraction Layer.
- Initialize and configure GPIO pins.
- Initialize the Serial Peripheral Interface.
- Set and clear a GPIO pin.
- Read and write data on the SPI.
- Set a delay on the hardware timer.
- Enable and disable interrupt requests.

Below we illustrate part of the Hardware Abstraction Layer code.

```
/* i.MX GPIO initialize structure. */
typedef struct _gpio_config
{
const char          *name;
__IO  uint32_t      *muxReg;
uint32_t             muxConfig;
__IO  uint32_t      *padReg;
uint32_t             padConfig;
GPIO_Type           *base;
```

```
uint32_t          pin;
} gpio_config_t;


/* Configuration parameters of the reset pin */
static gpio_config_t mrf24j40_gpio_reset_config = {
   "mrf24j40_nrst",          /* name */
   &IOMUXC_SW_MUX_CTL_PAD_SAI2_TX_BCLK,   /* muxReg */
   5,              /* muxConfig */
   &IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_BCLK,   /* padReg */
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_BCLK_PS(2) |  /* padConfig */
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_BCLK_PE_MASK |
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_BCLK_HYS_MASK,
   GPIO6,               /* base */
   20,                /* pin */
};

/* Configuration parameters of the irq pin */
static gpio_config_t mrf24j40_gpio_irq_config = {
   "mrf24j40_irq",           /* name */
   &IOMUXC_SW_MUX_CTL_PAD_SAI2_TX_DATA,   /* muxReg */
   5,              /* muxConfig */
   &IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_DATA,   /* padReg */
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_DATA_PS(2) |  /* padConfig */
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_DATA_PE_MASK |
   IOMUXC_SW_PAD_CTL_PAD_SAI2_TX_DATA_HYS_MASK,
   GPIO6,               /* base */
   22,                /* pin */
};

typedef struct EcspiState
{
   /* Pointer to ECSPI Transmit Buffer */
    uint8_t* txBuffPtr;
    /* The remaining number of bytes to be transmitted */
    uint8_t txSize;
   /* Pointer to ECSPI Receive Buffer */
    uint8_t* rxBuffPtr;
    /* The remaining number of bytes to be received */
    uint8_t rxSize;
    /* True if there is an active transfer */
    volatile bool isBusy;
} ecspi_state_t;

/* ECSPI runtime state structure */
static ecspi_state_t ecspiState;

/* Initialize the GPIO reset pin */
static void mrf24j40_gpio_init(void) {
gpio_init_config_t mrf24j40_gpio_reset_init_config = {
   .pin = mrf24j40_gpio_reset_config.pin,
   .direction = gpioDigitalOutput,
   .interruptMode = gpioNoIntmode
};

 GPIO_Init(mrf24j40_gpio_reset_config.base,
           &mrf24j40_gpio_reset_init_config);
```

```
}

/* Initialize the Hardware Abstraction Layer */
int8_t  mrf24j40_hal_init(void)
{
Hw_Timer_Init(HW_TIMER_A, NULL);
mrf24j40_gpio_init();
return 0;
}

/* Set a delay on the HW_TIMER_A */
void mrf24j40_hal_delay_us(uint16_t delay_us)
{
Hw_Timer_Action(HW_TIMER_A, delay_us);
}

/* Set the reset GPIO pin */
void mrf24j40_hal_resetn_high(void)
{
GPIO_WritePinOutput(mrf24j40_gpio_reset_config.base,
                    mrf24j40_gpio_reset_config.pin, gpioPinSet);
}

/* Clear the reset GPIO pin */
void mrf24j40_hal_resetn_low(void)
{
GPIO_WritePinOutput(mrf24j40_gpio_reset_config.base,
                    mrf24j40_gpio_reset_config.pin, gpioPinClear);
}

/* Initialize the Serial Peripheral Interface bus */
int8_t  mrf24j40_hal_spi_init(uint8_t port)
{
ecspi_init_config_t ecspiInitConfig = {
        .baudRate = 500000,
        .mode = ecspiMasterMode,
        .burstLength = ECSPI_MASTER_BURSTLENGTH,
        .channelSelect = BOARD_ECSPI_MASTER_CHANNEL,
        .clockPhase = ecspiClockPhaseSecondEdge,
        .clockPolarity = ecspiClockPolarityActiveHigh,
        .ecspiAutoStart = ECSPI_MASTER_STARTMODE
    };

ECSPI_Init(BOARD_ECSPI_MASTER_BASEADDR, &ecspiInitConfig);

return 0;
}

/* Read data from the SPI bus */
int8_t  mrf24j40_hal_spi_read(uint8_t *data, uint16_t len)
{
   return ECSPI_MasterTransfer(0, data, len);
}

/* Write data on the SPI bus */
int8_t  mrf24j40_hal_spi_write(uint8_t *data, uint16_t len)
{
   return ECSPI_MasterTransfer(data, 0, len);
}

/* Transmit and Receive an amount of data in no-blocking mode with
interrupt. This function is called within the
mrf24j40_hal_spi_read() and the mrf24j40_hal_spi_write() functions.
*/
static bool ECSPI_MasterTransfer(uint8_t* txBuffer, uint8_t*
rxBuffer, uint32_t transferSize)
{
    uint32_t len;

    if((ecspiState.isBusy) || (transferSize == 0))
    {
        return false;
    }

    /* Update the burst length to real size */
    len = (uint32_t)(transferSize * 8 - 1);
    ECSPI_SetBurstLength(BOARD_ECSPI_MASTER_BASEADDR, len);

    /* Configure the transfer */
    ecspiState.txBuffPtr = txBuffer;
    ecspiState.rxBuffPtr = rxBuffer;
    ecspiState.txSize = transferSize;
    ecspiState.rxSize = 0;

    /* Fill the TXFIFO */
    ECSPI_MasterTransmitBurst();
    /* Enable interrupts */
    ECSPI_SetIntCmd(BOARD_ECSPI_MASTER_BASEADDR,
                    ecspiFlagTxfifoEmpty, true);
    return true;
}

/* Fill the TXFIFO. */
static bool ECSPI_MasterTransmitBurst(void)
{
    uint8_t bytes;
    uint32_t data;
    uint8_t i;

    /* Fill the TXFIFO */
    while((ecspiState.txSize > 0) &&
        (ECSPI_GetStatusFlag(BOARD_ECSPI_MASTER_BASEADDR,
         ecspiFlagTxfifoFull) == 0))
    {
   /* first get unaligned part transmitted */
        bytes = ecspiState.txSize & 0x3;

        /* if aligned, then must be 4 */
        bytes = bytes ? bytes : 4;

        if(!(ecspiState.txBuffPtr))
        {
       /* half-duplex receive data */
            data = 0xFFFFFFFF;
        }
        else
        {
          data = 0;
          for(i = 0; i < bytes; i++)
              data = (data << 8) | *(ecspiState.txBuffPtr)++;
        }

        ECSPI_SendData(BOARD_ECSPI_MASTER_BASEADDR, data);
        ecspiState.txSize -= bytes;
        ecspiState.rxSize += bytes;
    }
    /* start transmission */
    ECSPI_StartBurst(BOARD_ECSPI_MASTER_BASEADDR);
    /* set transfer flag */
    ecspiState.isBusy = true;
    return true;
}

/* Receive data from RXFIFO  */
static bool ECSPI_MasterReceiveBurst(void)
{
    uint32_t data;
    uint32_t bytes;
    uint32_t i;

    while ((ecspiState.rxSize > 0) &&
      (ECSPI_GetStatusFlag(BOARD_ECSPI_MASTER_BASEADDR,
      ecspiFlagRxfifoReady) != 0))
    {
   /* read data from register */
        data = ECSPI_ReceiveData(BOARD_ECSPI_MASTER_BASEADDR);
        /* first get unaligned part received */
        bytes = ecspiState.rxSize & 0x3;
        /* if aligned, then must be 4 */
        bytes = bytes ? bytes : 4;

        /* not half-duplex transmit */
        if(ecspiState.rxBuffPtr)
        {
            for(i = bytes; i > 0; i--)
            {
                *(ecspiState.rxBuffPtr + i - 1) = data & 0xFF;
                data >>= 8;
            }
            ecspiState.rxBuffPtr += bytes;
        }
        ecspiState.rxSize -= bytes;
    }
    return true;
}

/* The interrupt service routine triggered by ECSPI interrupt */
void BOARD_ECSPI_MASTER_HANDLER(void)
{
    /* Receive data from RXFIFO */
    ECSPI_MasterReceiveBurst();
    /* Push data left */
    if(ecspiState.txSize)
    {
        ECSPI_MasterTransmitBurst();
        return;
    }

    /* No data left to push, but still waiting for rx data, enable
    receive data available interrupt. */
    if(ecspiState.rxSize)
    {
        ECSPI_SetIntCmd(BOARD_ECSPI_MASTER_BASEADDR,
                      ecspiFlagRxfifoReady, true);
        return;
    }

    /* Disable interrupt */
    ECSPI_SetIntCmd(BOARD_ECSPI_MASTER_BASEADDR,
                  ecspiFlagTxfifoEmpty, false);
    ECSPI_SetIntCmd(BOARD_ECSPI_MASTER_BASEADDR,
                  ecspiFlagRxfifoReady, false);

    /* Clear the status */
    ECSPI_ClearStatusFlag(BOARD_ECSPI_MASTER_BASEADDR,
                        ecspiFlagTxfifoTc);
    ECSPI_ClearStatusFlag(BOARD_ECSPI_MASTER_BASEADDR,
                         ecspiFlagRxfifoOverflow);

    ecspiState.isBusy = false;
}

/* Clear the interrupt status */
void mrf24j40_hal_irq_clean(void)
{
     GPIO_ClearStatusFlag(mrf24j40_gpio_irq_config.base,
          mrf24j40_gpio_irq_config.pin);
}

/* Disable GPIO pin interrupt */
void mrf24j40_hal_irq_disable(void)
{
     GPIO_SetPinIntMode(mrf24j40_gpio_irq_config.base,
          mrf24j40_gpio_irq_config.pin, false);
}

/* Enable GPIO pin interrupt */
void mrf24j40_hal_irq_enable(void)
{
     GPIO_SetPinIntMode(mrf24j40_gpio_irq_config.base,
          mrf24j40_gpio_irq_config.pin, true);
}
```

```
/* Interrupt handler */
void GPIO6_INT31_16_Handler(void)
{
     if (GPIO_IsIntPending(mrf24j40_gpio_irq_config.base,
           mrf24j40_gpio_irq_config.pin))
     {
          mrf24j40_hal_isr();
     }
}
```

5.4 Porting of the IEEE 802.15.4 stack

The IEEE 802.15.4 standard is the most suitable protocol for applications that implement low-rate wireless personal area networks. This protocol defines only the physical and the mac layers of the OSI model. Within this section we describe the structures and the primitive functions needed by the protocol stack. In FreeRTOS a protocol stack can be added to the application code through an add-on module, remembering to include the required files into the make file.

5.4.1 PHY and MAC service primitives

The services of a layer are the capabilities that it offers to the higher layer relying on the services proposed by the lower layer, following a sort of hierarchy. The information flow is exhibited through a set of instantaneous events, which indicate the provision of a service. Each event deals with passing a service between two layers, by means of a Service Access Point. A service primitive transmits the information by supplying the required service. These functions can be of four generic types:

- Request: it requests that a service is initiated.
- Indication: it indicates to the user an internal event.
- Response: it concludes a procedure that was previously invoked by an indication primitive.
- Confirm: it transmits the results of the associated previous service requests.

5.4.1.1 PHY layer

The physical layer performs the following tasks:

- Initialization (and switch off) of the radio transceiver.

- Energy detection, which means that this layer can detect the amount of power within the channel.
- Clear Channel Assessment.
- Setting the carrier frequency of the channel.
- Data transmission and reception.

This layer provides two services that can be accessed through two different Service Access Points: the PHY Data SAP and the PHY Management Service.

The physical layer data service (PD) deals with data transmission and reception. It includes three functions and it provides to the MAC layer an interface to the physical transmission medium. These functions manage all data exchanges between network devices. The functions are:

- uwl_PD_DATA_request: the upper layer uses this function to communicate to the physical layer the need to start a transmission.
- uwl_PD_DATA_confirm: after receiving a transmission request, the physical layer signals to the MAC layer the result of the transaction. This function is executed both in the case of transmission success and transmission error.
- uwl_PD_DATA_indication: this function is invoked at the time of a reception, it signals to the MAC layer the completion of the transmission.

The physical layer management entity (PLME) provides a set of functions that are exploited to handle the status and the parameters of the radio transceiver. The main functions are described below:

- uwl_PLME_CCA_request: it allows the MAC layer to request the beginning of the Clear Channel Assessment procedure, that will report the status of the transmission medium.
- uwl_PLME_CCA_confirm: once the CCA procedure is terminated, the physical layer signals the result to the upper layer. The transmission medium can be IDLE or BUSY.
- uwl_PLME_ED_request: it allows the MAC layer to request the beginning of the energy detection procedure within the radio channel.

- uwl_PLME_ED_confirm: once the procedure is terminated, the physical layer signals the energy detection result to the MAC layer.
- uwl_PLME_GET_request: it allows to request the value of a physical layer setting parameter. Examples of possible parameters are the transmission channel, the bitrate, the radio transceiver status.
- uwl_PLME_GET_confirm: this function communicates to the MAC layer the value of the requested parameter.
- uwl_PLME_SET_request: this function is used by the physical layer to set one of the characteristic parameters with a value that is passed to the function as input.
- uwl_PLME_SET_confirm: it signals to the upper layer if the setup operation is successfully ended.

5.4.1.2 MAC layer

The MAC layer manages the accesses and performs the following tasks:

- Generation of a network beacon, if the device acts as a coordinator.
- Synchronization to the network beacons, if the device is not a coordinator.
- Security support.
- Support to device association and disassociation.
- Implementation of the CSMA-CA protocol for the management of the accesses to the radio.
- Maintenance of network timing.

This layer can be divided into two main blocks: the MAC Common Part Sublayer and the MAC Layer Management Entity. Below we will describe the main functions of these two functional units.

The MAC Common Part Sublayer (MCPS) provides an interface for data exchange between the upper layer and the MAC sublayer. It supports data transmission and reception and it adds the ability to abort a running transmission. The main available functions are:

- uwl_MCPS_DATA_request: the upper layer uses this function to request (to the MAC layer) the start of a frame transmission.

- uwl_MCPS_DATA_confirm: the MAC layer signals to the upper layer the result of the transaction.
- uwl_MCPS_DATA_indication: the MAC layer notifies to the upper layer the conclusion of a frame reception.

The MAC Layer Management Entity (MLME) provides an interface to the upper layer for the management of the characteristic parameters of the MAC layer. The functionalities proposed by this sublayer can be summarized as follows:

- Device association and disassociation.
- Network beacon reception.
- Reading and writing PAN Information Base (PIB) attributes.
- Guaranteed Time Slots management.
- Notify if a device loses the connection with the coordinator.
- Supervise the operation time of the receiver.
- Channel scan.
- Management of the superframe configuration.
- Synchronization with a coordinator.
- Data request from a coordinator.
- Reset of the MAC.

We mention only some interesting functions:

- uwl_MLME_GET_request: this function is used to request the value of a particular parameter contained within the PAN Information Base.
- uwl_MLME_GET_confirm: the MAC layer answers to report the end of the request.
- uwl_MLME_SET_request: this function is used when the upper layer needs to set a MAC parameter.
- uwl_MLME_SET_confirm: it signals the end of the previous request.

5.4.2 Tasks

The protocol stack that we have ported within FreeRTOS derives from the uWireless library of Erika EE, that is another real time operating system. One of the greatest difficulties issued from the porting phase is the task management. Tasks in Erika are almost treated as simple functions, which have a beginning and an end. In Erika tasks must be registered, and to each task identifier is assigned the body of the function that must be executed after the task activation. The function that activate the task executes one instance of the task function and terminates the task execution. If we want to rerun the task function, we need to call again the activate task function. If we need to periodically iterate the task, we had to use the alarm notifications, which reactivate the task after a period is expired. The period duration is measured in timer ticks.

In FreeRTOS, as we have seen in chapter 3, the implementation of a task is a bit different. A task is a program with an entry point that run forever within an infinite loop, without exiting, until the task is deleted. We have decided to follow the Erika EE task conception, adopting a similar solution. Since we have already implemented a timer, that always starts when the stack is initialized, we develop a solution consisting of a single task. We register some callback functions (that in Erika were called tasks) into structures, so that, at each timer tick, the main task calls a function, which checks if there are any callbacks that need to be executed. We therefore decide to define the tasks that we had seen in Erika, as well as trivial functions. Within Erika, when we decide to activate a task, if it is a periodic task, we must define the offset (how long before the task will be activated) and the period. In FreeRTOS, we have chosen to use an array of structures to achieve this purpose. Every structure contains:

- The status of the task (active or inactive)
- A numeric value that represents a counter which indicates how many timer ticks remain before we had to call the function
- The period (if needed) that indicates how often the function must be called
- A pointer to the function that must be executed.

At every timer tick, we scan the array of structures. For each element of the array, the counter is checked. If the counter equals to zero, the corresponding callback must be executed. If the counter is non-zero, it is decremented by one unit. If the counter is zero and the callback function is periodic, in addition to execute the callback code, we must reset the counter, assigning to it a value equal to the period. If the task is not periodic, after the callback execution, its status is set to inactive. These functions (tasks in Erika) are called mainly for:

- receive a PHY frame
- process a received beacon
- process a received data
- process a received MAC command
- manage the begin of a timeslot.

Below we illustrate the code that we have implemented to manage tasks.

```
volatile uint32_t uwl_kal_time_counter = 0;

/* Function identifiers */
enum {
   MAC_BACKOFF_PERIOD,
   MAC_BEFORE_TIMESLOT,
   MAC_GTS_SEND,
   MAC_PROCESS_RX_BEACON,
   MAC_PROCESS_RX_COMMAND,
   MAC_PROCESS_RX_DATA,
   MAC_TIMESLOT,
   PHY_READ_DISPATCHER,
   TASK_ID_MAX,
};

/* Mutex semaphores that are used within tasks */
SemaphoreHandle_t MAC_MUTEX = NULL;
SemaphoreHandle_t MAC_SEND_MUTEX = NULL;
SemaphoreHandle_t MAC_GTS_SEND_MUTEX = NULL;
SemaphoreHandle_t MAC_RX_BEACON_MUTEX = NULL;
SemaphoreHandle_t MAC_RX_COMMAND_MUTEX = NULL;
SemaphoreHandle_t MAC_RX_DATA_MUTEX = NULL;

typedef void (*uwl_task_func)(void);

/* Task structure */
typedef struct {
   uint16_t status;
   uint16_t offset;
   uint16_t period;
   uwl_task_func func;
} uwl_task_struct;

/* The array of task structures */
uwl_task_struct uwl_task_list[TASK_ID_MAX];
```

```c
/* Possible task status*/
#define UWL_TASK_ACTIVE 1
#define UWL_TASK_INACTIVE 0

/* Timer action. At every timer tick, this function scans the array and,
if the offset of a callback is zero, it executes the callback. It also
updates the offset, or it set the status to inactive. If the offset is
not zero, the offset is decremented by one. */
void uwl_kal_external_timer_action(void)
{
   uint16_t i;

   uwl_kal_time_counter++;

   for (i = 0; i < TASK_ID_MAX; i++) {
      if (uwl_task_list[i].status == UWL_TASK_INACTIVE)
         continue;
      if (uwl_task_list[i].offset == 0) {
         uwl_task_list[i].func();
         if (uwl_task_list[i].period)
            uwl_task_list[i].offset = uwl_task_list[i].period;
         else
            uwl_task_list[i].status = UWL_TASK_INACTIVE;
      } else {
         uwl_task_list[i].offset--;
      }
   }
}

/* Get the value of the time counter */
uint32_t uwl_kal_get_time(void)
{
   return uwl_kal_time_counter;
}

/* Initialization */
int8_t uwl_kal_init(uint32_t tick_duration)
{
   static int first_call = 1;

   if (first_call) {
      first_call = 0;
      MAC_MUTEX = xSemaphoreCreateMutex();
      MAC_SEND_MUTEX = xSemaphoreCreateMutex();
      MAC_GTS_SEND_MUTEX = xSemaphoreCreateMutex();
      MAC_RX_BEACON_MUTEX = xSemaphoreCreateMutex();
      MAC_RX_COMMAND_MUTEX = xSemaphoreCreateMutex();
      MAC_RX_DATA_MUTEX = xSemaphoreCreateMutex();
   }

   if (tick_duration != 0) {
      Hw_Timer_Init(HW_TIMER_B, uwl_kal_external_timer_action);
      Hw_Timer_Action(HW_TIMER_B, tick_duration);
   }
   return 1;
}
```

```c
/* Activate a task. The offset and the period are equal to zero, so the
callback is executed only one time */
void uwl_kal_activate(uint16_t task_id)
{
  if (task_id >= TASK_ID_MAX)
    return;
  uwl_task_list[task_id].offset = 0;
  uwl_task_list[task_id].period = 0;
  uwl_task_list[task_id].status = UWL_TASK_ACTIVE;
}

/* Set the status of a task to inactive */
void uwl_kal_cancel_activation(uint16_t task_id) {
  if (task_id >= TASK_ID_MAX)
    return;
  uwl_task_list[task_id].status = UWL_TASK_INACTIVE;
}

/* Set the body of a task. When a task is activated, the callback that
corresponds to the task body is executed */
int uwl_kal_set_body(uint16_t task_id, uwl_task_func task_func)
{
  if (task_id >= TASK_ID_MAX)
    return -1;
  uwl_task_list[task_id].func = task_func;
  return 0;
}

/* Activate a periodic task, setting the offset and the period.*/
void uwl_kal_set_activation(uint16_t task_id, uint16_t offset, uint16_t
period)
{
  if (task_id >= TASK_ID_MAX)
    return;
  uwl_task_list[task_id].offset = offset;
  uwl_task_list[task_id].period = period;
  uwl_task_list[task_id].status = UWL_TASK_ACTIVE;
}
```

Conclusions

The main goal of this thesis is to prepare an environment that can be used to create a wireless sensor network that guarantees low costs and low consumption, since these types of network are widespread in the IoT world. We have seen that the IEEE 802.15.4 protocol is the most suitable for networks that do not require high data-rates, especially in its beacon enabled version with guaranteed time slots. GTSs are very useful because they allow to assign portions of a superframe to a device, assuring an exclusive access to the channel to communicate with the network coordinator. We have illustrated how the heterogeneity of an Asymmetric Multi Processing system fulfils the need for both real-time responsiveness and processing power. We showed the characteristics of the FreeRTOS real-time operating system, explaining that we choose it because it is simple to use, it is one of the most used open source OS within embedded systems and it supports a lot of hardware architectures. In this work we made the porting of the MRF24J40 device driver and of the protocol stack within FreeRTOS, using as hardware platform the VAR-SOM-MX7 board, that supports AMP. We had primarily to develop the Hardware Abstraction Layer of the driver and manage the tasks of the protocol stack within FreeRTOS. These were the two more complex phases of the software development.

The implemented software can be exploited for the development of a future IoT application that manages a low-rate wireless personal area network. The VAR-SOM-MX7 board, equipped with FreeRTOS, will be integrated within a network, acting as coordinator node. A sensor network could send data to the Cortex-M4 core, which can process data in real-time before passing them to the Cortex-A7 core. The AMP technique will allow to develop complex algorithms on the Linux side, for the processing of the data information transmitted by the sensors. Data could also be maintained within the cloud. In addition, a mobile application could be developed to allow a remote control over the sensor network. Possible high-level applications that could employ this architecture may be a system that manages the temperature or the lighting of a building, an intruder detection system or a system that controls the entertainment devices in a home.

References

[1] Chiara Buratti, *Performance Analysis of IEEE 802.15.4 Beacon-Enabled Mode*, 2010.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5378538`

[2] Yushi Uematsu; Kentaro Kobayashi; Hiraku Okada; Masaaki Katayama, *A Study on Multiple Access Schemes for Wireless Control over the IEEE 802.15.4 Beacon-Enabled Mode*, 2017.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7925825`

[3] Vidya Honguntikar; Gangadhar S. Biradar, *Performance Analysis of GTS Allocation in IEEE 802.15.4 with CSMA/CA for Wireless Sensor Network*, 2015.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7562881`

[4] IEEE Standard for Low-Rate Wireless Networks.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7460875`

[5] IEEE 802.15.4 Stack User Guide.
`https://www.nxp.com/docs/en/user-guide/JN-UG-3024.pdf`

[6] Mastering the FreeRTOS Real Time Kernel.
`https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf`

[7] VAR-SOM-MX7 datasheet.
`http://www.variscite.com/images/stories/DataSheets/VAR-SOM-MX7/VAR-SOM-MX7_v1_X_VAR-SOM-MX7-5G_datasheet_v1_6.pdf`

[8] VAR-MX7 Custom Board Datasheet.
`http://www.variscite.com/images/stories/DataSheets/VAR-SOM-MX7/VAR-MX7CustomBoard_Datasheet_V_1_X.PDF`

[9] MRF24J40 Datasheet.
`http://ww1.microchip.com/downloads/en/DeviceDoc/39776C.pdf`

[10] MRF24J40MA/MB PICtail/PICtail Plus Daughter Board User's Guide.
`http://ww1.microchip.com/downloads/en/DeviceDoc/51867A.pdf`

[11] VAR-SOM-MX7 User Guide.
`http://variwiki.com/index.php?title=VAR-SOM-MX7`