

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Wireless Sensor Network for pH and Dissolved Oxygen Measurement for Bioprocess Monitoring

Daniel Grossi Zurita

MASTER IN ELECTRICAL AND COMPUTERS ENGINEERING

UMBC Supervisor: Professor Yordan Kostov

FEUP Supervisor: Professor Paulo Portugal

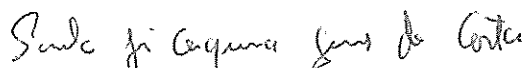
July 27, 2014

A Dissertação intitulada

“Wireless Sensor Network for pH and Dissolved Oxygen Measurement for
Bioprocess Monitoring”

foi aprovada em provas realizadas em 18-07-2014

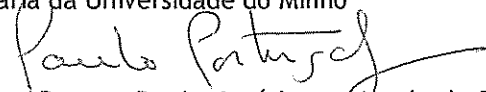
o júri



Presidente Professor Doutor Paulo José Cerqueira Gomes da Costa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

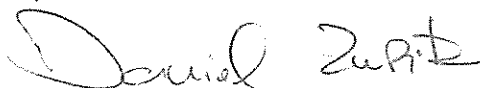


Professora Doutora Filomena Maria da Rocha Menezes de Oliveira Soares
Professora Associada do Departamento de Electrónica Industrial da Escola de
Engenharia da Universidade do Minho



Professor Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua
exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente
autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou
inspirados em trabalhos de outros autores, e demais referências bibliográficas
usadas, são corretamente citados.



Autor - Daniel Grossi Zurita

Faculdade de Engenharia da Universidade do Porto

Resumo

Recentes desenvolvimentos na área tecnológica têm levado a que as indústrias químicas e biológicas melhorem cada vez mais a monitorização e o controlo dos seus processos. No entanto, existem ainda alguns aspetos que devem ser revistos a fim de otimizar ainda mais os métodos de deteção, computação e comunicação, tornando-os melhores e mais eficazes.

Os processos biológicos que ocorrem dentro de culturas celulares, podem ser monitorizados e controlados pela observação da evolução de alguns de parâmetros. Dois destes parâmetros são o pH e o oxigénio dissolvido, e são medidos utilizando diferentes tipos de sensores, como eléctrodos de vidro ou ópticos. Uma das desvantagens relacionadas com este tipo de monitorização é a possibilidade de as células em crescimento se infiltrarem nos sensores, comprometendo assim os resultados das suas leituras. Além disso, alguns tipos de sensores invasivos podem comprometer a esterilidade. Numa rede de sensores com fios, as medições podem ser comprometidas devido à presença de dióxido de carbono dentro das incubadoras que contêm as culturas de células.

Esta tese visa procurar uma solução para os problemas acima mencionados através do desenvolvimento de uma rede de sensores sem fios. Esta deve comunicar com os sensores no interior da incubadora, evitando que as medições sejam comprometidas. A rede é composta por três níveis. O nível mais baixo é composto por *sensor boards* e é responsável pela execução de tarefas. Por sua vez, o segundo nível é composto por uma *base station* e um computador, sendo responsável por solicitar às *sensor boards* a execução das tarefas bem como a posterior aquisição dos dados resultantes das mesmas. Finalmente, uma interface visual foi concebida para facilitar a interação do utilizador com a rede.

Abstract

The latest developments in monitoring and control technology systems have benefited the biological and chemical industries and improved their associated processes. However, there are still some issues that are to be overcome in order to further optimize these methods of sensing, computing, and communication leading to better and more effective monitoring and control.

Various bioprocesses occur within cell cultures that can be supervised by observing and overseeing the changes in a number of parameters. Two of these are pH and dissolved oxygen, which are measured using different types of sensors like glass electrodes or optrodes (optical electrode). One of the issues related to this type of monitoring is the possibility of the growing cells infiltrating the sensors, therefore invalidating the results. Also, some types of sensors, if invasive, can compromise sterility. In a wired sensor network, the presence of carbon dioxide in incubators containing the cell cultures can also compromise the measurements.

This thesis is concerned with solving the problems mentioned above by developing a wireless sensor network to communicate with the sensors inside the incubator, thus, preventing compromising the measurements. The network consists of three levels. One that is aimed for operation purposes, which is achieved by a sensor board. The second level joins one base station and a personal computer. This level is responsible for requesting and acquiring data to and from the sensor board. Finally, a visual interface was designed to support the user when interacting with the network.

Acknowledgments

It is with great pleasure that I express my deep and sincere gratitude to all the people that, either professionally or personally, directly or indirectly, made the completion of this work possible.

I would like to thank Professor Yordan Kostov, for the availability, attention paid and professionalism in the construction of this project, and even the unique opportunity to develop this dissertation joining his fantastic laboratory team. I'm thankful to Professor Paulo Portugal, for his active involvement in the structure and organization of the reviewed topics, as well as the shared knowledge.

A special acknowledgment to Neha Sardesai, not only by the immense support and motivation given so many times, but also for her extreme patience. I would also like to thank Ricardo Ribeiro who has always stood by my side in the various ups and downs I faced throughout my academic life.

I would like to express my sincere gratitude to my parents, the ones responsible for the person I am today. I thank my father for all his guidance and unconditional support. To my mother, for her love and affection. You are definitely my biggest inspiration.

Dedicated to my uncle and to my grandmother.

Daniel Grossi Zurita

Contents

1	Project Overview	1
1.1	Introduction	1
1.1.1	Background and Motivation	1
1.2	Related Theory for Protocol Development	2
1.2.1	Bioprocess Monitoring	2
1.2.2	Wireless Sensor Networks	5
1.2.3	Related Work	12
1.3	Structure of the Thesis	13
2	System Design	15
2.1	System Requirements	15
2.2	Structure Breakdown	15
2.2.1	Lower Level: Sensor Board	16
2.2.2	Middle Level: Base Station and Personal Computer	18
2.2.3	Higher Level: Visual Interface	19
3	Wireless Network Protocol	21
3.1	Network Setup	21
3.2	Communication from the Computer to the Base Station	23
3.3	Two Way Communication Between Base Station and Sensor Board	25
3.3.1	Processing Commands 1, 2, and 3	25
3.3.2	Processing Command 4	26
3.3.3	Processing Command 5	26
3.4	Communication from the Base Station to the Computer	26
3.5	Network Operation with Two Sensor Boards	26
4	Results	29
4.1	Code Description	29
4.1.1	Base Station Code	29
4.1.2	Sensor Board Code	34
4.2	Tests	38
4.2.1	Efficiency	38
4.2.2	Visual Interface	40
5	Conclusions	41
5.1	Discussion	41
5.2	Future Work	42

A	Board Schematics	43
A.1	Base Station Schematic	43
A.2	Sensor Board Schematic	44
B	Programming Codes	45
B.1	IAR Code	45
B.1.1	Base Station Code	45
B.1.2	Sensor Board Code	58
B.2	Include Files	66
B.2.1	stdint.h	66
B.2.2	include_ez430.h	68
B.3	LabVIEW Interface	72
B.3.1	Block Diagram	72
	References	75

List of Figures

1.1	Wireless Sensor Network.	5
1.2	Wireless Network Topologies.	6
1.3	Short-Range Wireless Networking Classes [1].	7
1.4	WirelessHART Mesh Networking [2].	9
1.5	SimpliciTI protocol architecture [3].	10
1.6	20 Series Wireless by InsiteIG.	12
2.1	System architecture block diagram.	16
2.2	Sensor board block diagram.	16
2.3	Sensor board top and bottom views.	17
2.4	Base Station board block diagram.	18
2.5	Base station top and bottom views.	19
2.6	LabVIEW visual interface.	19
3.1	Request frame.	21
3.2	Request frame example.	22
3.3	Grant permission frame.	22
3.4	Grant-permission frame example.	22
3.5	Network's state machine.	23
3.6	Command 2 Instruction frame.	24
3.7	Command 4 Instruction frame.	24
3.8	Command frame.	25
3.9	Command 4 frame example.	25
3.10	Command 1 frame for broadcasting.	27
4.1	Function <i>Main()</i> implemented in the Base Station.	29
4.2	Interrupt Service Routine for radio events implemented on the Base Station.	30
4.3	Function <i>RFReceivePacket()</i>	31
4.4	Function <i>Send_Network_Permission()</i>	32
4.5	Function <i>Send_Command()</i>	33
4.6	Cycle that uploads the data payload to the Personal Computer.	33
4.7	Function <i>Join_Network()</i>	34
4.8	Interrupt Service Routine for radio events implemented on the Sensor Board.	35
4.9	Functions implemented to enable the red LED.	36
4.10	Functions <i>Setup_Flash_Clock()</i> and <i>Write_Flash_SegC()</i>	37
4.11	Functions <i>Send_Data()</i>	37
4.12	Incubator used for tests.	38
4.13	Demonstration of the execution of command 4 followed by command 5 in the Visual Interface.	40

A.1	Base station schematic.	43
A.2	Sensor board schematic.	44
B.1	LabVIEW block diagram that acquires data via USB.	72
B.2	LabVIEW block diagram that interfaces the Base Station with the Personal Computer.	73

List of Tables

1.1	Main characteristics of the different sensors.	4
1.2	Advantages and disadvantages of topologies.	6
1.3	Protocol comparison [4, 5].	11
3.1	List of commands and its descriptions.	24
4.1	Current consumption for output levels of the RFC [6].	39
4.2	Communication range for different scenarios.	40

Acronyms and Abbreviations

ADC	Analog-to-Digital Converter
BLE	Bluetooth Low Energy
BS	Base Station
DAC	Digital-to-Analog Converter
DO	Dissolved Oxygen
DR	LED Drive
FFD	Full-Function Device
ISFET	Ion-Sensitive Field Effect Transistor
ISR	Interrupt Service Routine
MAC	Media Access Control
PC	Personal Computer
PHD	Photodetector Circuit
RFC	Radio Frequency Chip
RFD	Reduced-Function Device
SB	Sensor Board
SPI	Serial Peripheral Interface
TI	Texas Instruments
UART	Universal asynchronous receiver/transmitter
USCI	Universal Serial Communication Interface
VI	Visual Interface
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network
μ C	Microcontroller

Chapter 1

Project Overview

1.1 Introduction

1.1.1 Background and Motivation

Environmental and bioprocess monitoring are some of the fields that have most benefited from the evolution of sensors, in particular bio-sensors, and sensor networks. They represent a group of applications that provide great development for scientific communities [7].

Throughout the years it has become possible to measure parameters as pH and Dissolved Oxygen (DO) on biologically derived materials such as microorganisms, enzymes and mammalian cell cultures. Monitoring mammalian cell cultures is essential considering that some of the parameters measured allow scientists to evaluate their growth, rate and cessation, and metabolism. These parameters are usually measured with electrodes (probes) or sensor patches, each one with their liability (i.e., compromised sterility, change of calibration due to sterilization, failure of the sensor in the process of sterilization, etc.), and they show information from the biological component after being transformed into a signal which can be more easily quantified. Combining sensing, computation and communication, this type of technology can be used to monitor environments that require special attention given that the tiniest variation of any kind of physical property may represent important data.

One of the biggest problems in this field is the presence of carbon dioxide inside incubators [8], that contain the cell cultures to be monitored, and that represents a liability when using a wired sensor network. Based on this premise, the smartest way to surpass this problem is implementing a wireless sensor network (WSN).

A common WSN, among other specifications, requires a communication protocol that may be selected from available standards including 2.4 GHz radios based on either IEEE 802.15.4 or IEEE 802.11 (Wi-Fi) standards or proprietary radios, which are usually 900 MHz [9]. One of the designing challenges of this implementation is to create a low-power solution, and the ones to be considered are those based on IEEE 802.15.4, for example, ZigBee, WirelessHART, Bluetooth Low Energy (BLE), and SimpliciTI. When opting for WSNs, there are several ways of interaction,

for data-collection, that should be analyzed. This crucial step can be triggered, for example, by event detection or periodically [10].

According to that, the purpose of this project is to develop a WSN for pH and DO monitoring. The deployment of a network of this nature brings advantages in many ways, from wire reduction, which represents a cost reduction, to robustness, scalability and the data decentralization, circumventing the problem of linking failures. Another advantage is that there are no wires going through the access ports or door, minimizing the risk of contamination of the incubators and decreasing the release of CO₂ to the lab environment.

1.2 Related Theory for Protocol Development

1.2.1 Bioprocess Monitoring

A bioprocess uses organisms, tissues, cells, or their molecular components to act on substrates, and obtain desired products from them. These bioprocesses usually occur inside a bioreactor which is a vessel that holds a biologically active environment. Penicillin antibiotics are a good example of substances that are produced using the method described above.

A proper control and monitoring of this process is required to optimize the development of desired output products, not just in pharmaceuticals, but also in bio-diesels, biodegradable plastics, etc. This type of monitoring tracks important variables that give important information about the cells health and growth rate. Two of the most common parameters that are measured and controlled are pH and DO.

The pH content of a solution is defined as the measurement of acidity or basicity of the solution [11]. This parameter measures the hydronium ion concentration and it is mathematically defined as:

$$pH = -\log_{10}[H_3O^*] \quad (1.1)$$

pH values range from 0, which indicates a high concentration of hydronium ions, detecting an acid solution, to 14, that corresponds to the opposite, in other words, indicating a low concentration of hydronium ions, referring to basic solutions. Substances with a pH of 7 are considered to be neutral.

In biotechnology, more precisely in cell culture experiments, there is a need of getting constant information about this parameter, but there are factors that can compromise the accuracy of conventional online pH measurements, such as drift and fouling [11]. They can be caused by bioprocesses containing proteins or chemical changes that interfere with the precision of the readings.

Thus, optimal cell growth depends heavily on the tight control of pH, which is the most commonly measured parameter [12]. It can be concluded that, it is crucial that neither the sensors nor the measurement methods introduce any type of contamination to the process.

The most common pH sensor consists of two parts, a glass electrode and a reference electrode, and the electric potential created between them is a function of the pH value of the measured solution [13]. The value of pH is calculated using the Nernst equation and the potential difference values obtained before:

$$E = E_0 + k \times T \times \log_{10}[H_3O^*] \quad (1.2)$$

The $k \cdot T$ being the Nernst factor and using the equation 1.1:

$$pH = \frac{E_0 - E}{k \times T} \quad (1.3)$$

A similar instrument is the pH meter [14], which has a measuring probe (a glass electrode) connected to an electronic meter that measures and displays not the potential, but the pH value after conversion.

Some of the disadvantages of this two methods are listed below [11]:

- Requires monitoring by technicians.
- Requires recalibration by technicians.
- Calibration drifts over time.
- Drift or fouling can affect accuracy.
- Unsuitable for online application: must remove probe from process or develop external measurement protocol to recalibrate.

These disadvantages led to the development and manufacturing of alternative technologies for applications where the methods above were not optimal.

Ion-Sensitive Field Effect Transistor (ISFET) is a non-glass solution similar to the glass electrode but with a different principle of operation which is based on the control of the current flow between two semiconductor electrodes. Like the above, this electrode can be described by the equation 1.2. Despite the fact that ISFET sensors measure more quickly and are less temperature dependent than glass electrodes, they require calibration and are susceptible to biofouling. This is not a flawless method, therefore, it is not a method of choice for bioprocessing pH measurement [11, 15].

Another existing solution, and an important one in biotechnology, is the usage of an optical sensor, which has a photodetector to measure the change in fluorescence of an exposed fluorophore indicator as a function of pH. This technique is based on emission and reflection of a beam of light. First, a pulse of light is emitted to a pH membrane which changes color according to the measured sample. The role of the photodetector is to measure the reflected light pulse. After interacting with a fluorophore indicator, the modified optical signal is guided, via fiber, to a photodetector, enabling the measurement of pH remotely without the need for constant recalibration. It should be

noted that the measurements must take place inside an incubator, since they are sensitive to light [11, 16, 17].

In the last few years, some companies have developed solutions that, unlike the ones presented so far, are non-invasive and they solve the problem of accuracy, sterility, recalibration, drift and biofouling. There are two options, sensor spots mounted in transparent vessels and amperometric sensors. Table 1.1 resumes some of main characteristics of the sensors reviewed.

Table 1.1: Main characteristics of the different sensors.

	Glass Electrodes	Optical Sensors	ISFET Sensors	Sensor Spots	Amperometric Sensors
pH Range	0-14	5-9	0-14	5.5-8.5	0-14
Subject To Biofouling	Yes	Yes	Yes	—	No
Calibration Free	No	No	No	Yes	Yes
Compromise Sterility	Yes	No	Yes	No	No
Requires Special Monitoring	Yes	Yes	Yes	No	No

As mentioned before, another important variable to be monitored is the DO. This represents a relative measure of the amount of oxygen that is dissolved in a given medium and it is expressed as a concentration in milligrams per liter (mg/L) or parts per million (ppm). Theoretically, the amount of DO in a solution is dependent on three factors, the temperature and the salinity, which are inversely proportional to oxygen solubility, and the atmospheric pressure which is directly proportional to DO.

Just like pH, DO can be measured, in aqueous solutions, with a probe/meter such as an oxygen sensor or an optrode and this measurement is important because it gives information about the health of the organisms.

The most used oxygen sensor is the Clark-type electrode, which consists in two parts, a cathode and an anode, both submerged in an electrolyte. Oxygen is diffused, through a membrane, to the sensor and is reduced at the cathode, therefore creating electrical current. This current is proportional to the oxygen concentration. A relevant fact of these sensors is that they consume O_2 once they are active and this can influence the measurements negatively.

On the other hand, there is a sensor based on optical measurements called optrode. The principle consists of a chemical film that is glued to the tip of an optical cable. When O_2 molecules collide with this film, the photoluminescence is quenched. Once there is no consumption of O_2 during the process, this solution offers an important asset to DO sensing [18].

These bioprocesses occur in a temperature controlled environment inside an incubator. The fact that the pH/DO sensors are wired proves to be a hassle while handling the equipment. The presence of wires in the incubator compromises the sterility of the environment. Crucial setbacks like the ones mentioned lead to the development of an integrated system that not only provides the measurements of these parameters, but also works wirelessly, therefore operating as a stand-alone device.

1.2.2 Wireless Sensor Networks

The advance in several fields of knowledge like, miniaturization and cost reduction of computers, wireless communication, sensor design and energy storage, has contributed to the development of low-cost, low-power, multifunctional sensor nodes untethered in short distances [19]. Nowadays, the concept of WSN appears in a wide range of applications from objects monitoring (such as machines and buildings), seismic detection and military programs, to habitat, environmental and bioprocesses monitoring [20].

A WSN has two types of nodes, the sensor nodes and the Base Station (BS), as shown in Figure 1.1. Each sensor node has several parts: a radio transceiver, a microcontroller, a data acquisition circuit to interact with the sensors, and an energy source, usually a battery [21]. These nodes, that can vary in quantity from a few units to a few thousands, depending on the BS, are distributed spatially, whether inside the environment under observation, or nearby. On the other hand, the BS receives information previously forwarded from the other nodes.

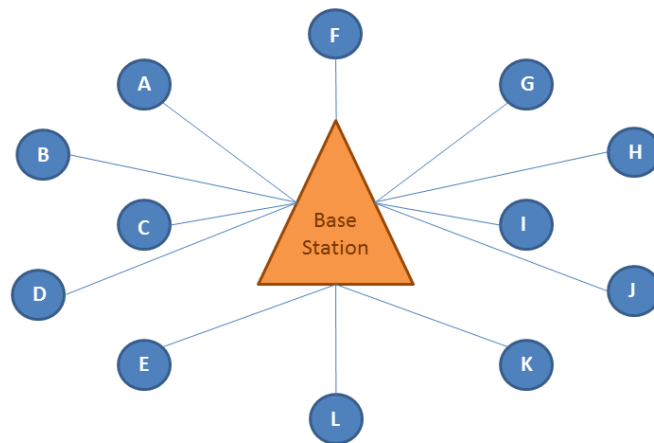


Figure 1.1: Wireless Sensor Network.

In wireless networks, one of the main requirements is the determination of the network topology which can be separated in two categories, physical and logical. Physical topology determines the way that the many devices are arranged while the logical topology is responsible for the communication between each other regardless of the physical topology [20, 22].

Star topology is a point-to-point architecture where the nodes communicate directly with the BS [22]. The fact that topologies like this have direct wireless connections, makes it a low-power solution, a critical requirement these days. However, communication between nodes can only be made by passing data through the BS, thus representing a drawback in some cases.

If connected over a mesh topology, the nodes can communicate with each other, as long as they are within reach. It is used in cases where the range distance is bigger than 100 meters since the nodes pass data over each other decreasing the possibility of data loss.

However, there is a special case that blends both star and mesh networks, called cluster-tree topology. Devices are grouped around routers that communicate with each other. It follows the next principle of operation:

- The first device starts the connection and becomes the root of the tree.
- Other nodes can join the network at the root or at others points.
- Every device is aware of their "parent" and "child" nodes.

This hierarchy reduces the routing complexity. Figure 1.2 illustrates the architectures described above.

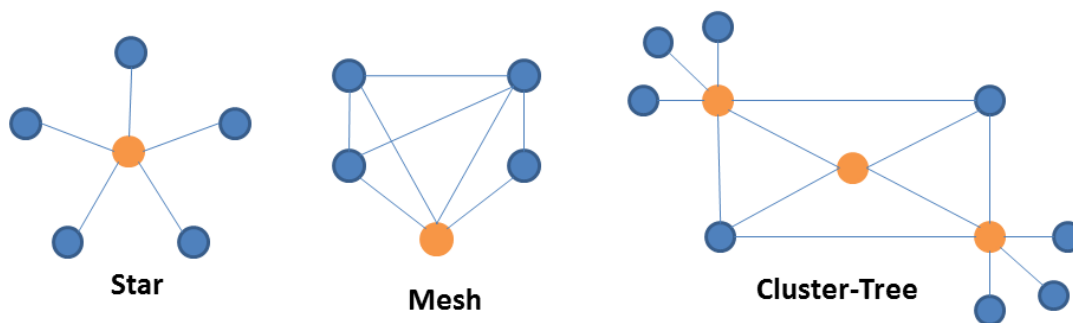


Figure 1.2: Wireless Network Topologies.

Despite the possibility of building a network without understanding the difference between topologies, it is important to become familiar with them, thus, in order to better understand their assets, Table 1.2 summarizes the advantages and disadvantages of each one.

Table 1.2: Advantages and disadvantages of topologies.

Topology	Advantages	Disadvantages
Star	<ul style="list-style-type: none"> - Simplicity - Centralized systems - Low latency and high bandwidth - Low power consumption 	<ul style="list-style-type: none"> - Central node dependent - Limited spatial cover - Single point to failure - Inefficient slave-to-slave communication
Mesh	<ul style="list-style-type: none"> - Distributed processing - Large spatial coverage - Scalable - Low/medium complexity 	<ul style="list-style-type: none"> - Complexity of routing - High latency and low bandwidth
Cluster-tree	<ul style="list-style-type: none"> - Large spatial coverage - Many nodes possible - Medium complexity 	<ul style="list-style-type: none"> - Low reliability - High latency and low bandwidth - Asymmetric power consumption

How devices communicate with each other is an important feature of a network and when referring to wireless solutions there are a number of protocols which satisfy several specific requirements. A family of standards, IEEE 802, is responsible for local and metropolitan area networks, and is divided in many working groups. The most common used standards are the *Ethernet*, which is wired, the Wireless Local Area Network (WLAN) and the Wireless Personal Area Network (WPAN) [23]. The working group associated with the implementation of a WSN is the IEEE 802.15, divided in seven task groups, like Bluetooth, high and low-rate WPAN, body area network among others, as seen in figure 1.3.

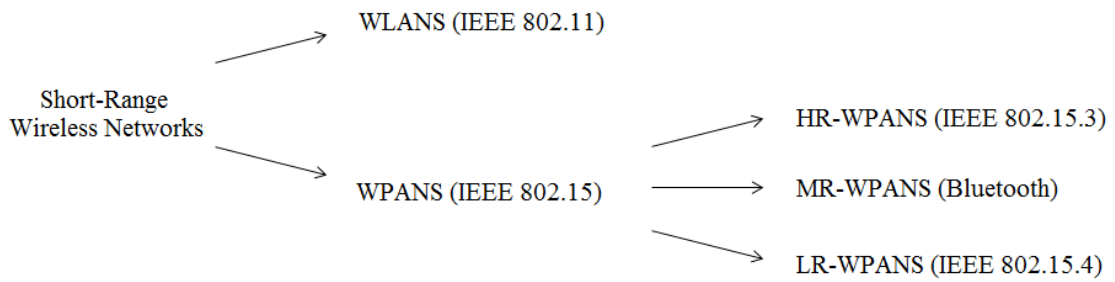


Figure 1.3: Short-Range Wireless Networking Classes [1].

IEEE 802.15.4 specifies the lower protocol layers: the physical layer and the media access control (MAC). There is a wide range of products and application areas based on this standard - safety, security and control of homes, mobile services within business and healthcare, industrial automation, home entertainment, environmental and habitat monitoring among others [24]. The physical layer provides the data transmission service through the RF transceiver, which can work in three frequency bands, 868.0-868.6 MHz for the European band, 902-928 MHz for the North American band and 2400-2483.5 MHz worldwide. As for the MAC layer, it handles the data service and controls frame validation, time slots and node associations. The way how the network is deployed is also specified by this standard and there are two different types of nodes, Full-function Devices (FFD) used as BS or common nodes and Reduced-function Devices (RFD) which can only be used as common nodes, due to their simplicity.

It is important to make the distinction between protocols as they serve different purposes. Hence, some of the most important protocols that refer to the low-rate WPANs group based on IEEE 802.15.4 standard will be discussed next.

ZigBee is a Radio Frequency (RF) communication based on IEEE 802.15.4 standard and is a protocol that is usually implemented in applications with low data rate, long battery life, and secure networking. It operates in the 2.4GHz frequency (there are also modules optimized for low frequency) and has a defined rate of 250 kbit/s, best suited for periodic or intermittent data or a single signal transmission from a sensor or input device. The low duty cycle makes it low power as the node can be, most of the time, sleeping. A ZigBee-based network supports a maximum number of nodes of 1024 within a 200 meters range in straight line and free of obstacles [25].

This specification supports the common star and tree topologies but though low-powered, ZigBee devices are able to transmit data over long distances by passing data through other nodes, therefore creating a mesh network. In star topology, the network is controlled by the BS, responsible for initiating and maintaining the devices on the network, and on the other hand, further nodes can communicate directly with the base. As for mesh and tree topologies, the BS is responsible for initiating the network and choosing some network parameters, with the possibility for the network to be extended through the use of routers [24].

The low installation and running cost provided by ZigBee helps counter the expensive and complex architecture problems with existing systems.

WirelessHART is a network protocol built over a mesh topology that includes three critical elements:

- Wireless field devices connected to process or plant equipment.
- Gateways that enable communication between these devices and host applications connected to a high speed backbone or other existing plant communications network.
- A network manager that is responsible for configuring the network, scheduling communications between devices, managing message routes and monitoring network health. It can be integrated into the gateway, host application, or process automation controller.

As stated in [26], the network manager determines the redundant routes based on latency, efficiency and reliability. To ensure the redundant routes remain open and unobstructed, messages continuously alternate between the redundant paths. Consequently, like the Internet, if a message is unable to reach its destination by one path, it is automatically re-routed to follow a known-good, redundant path with no loss of data.

Each device in the mesh network doesn't have to communicate directly to a gateway, but just forward its message to the next closest device. This extends the range of the network and provides redundant communication routes to increase reliability.

The mesh design of this protocol, which can be found illustrated in figure 1.4, also makes adding or moving devices easy. As long as a device is within range of others in the network, it can communicate.

For flexibility to meet different application requirements, The WirelessHART standard supports multiple messaging modes including one-way publishing of process and control values, spontaneous notification by exception, ad-hoc request/response, and auto-segmented block transfers of large data sets. These capabilities allow communications to be tailored to application requirements thereby reducing power usage and overhead.

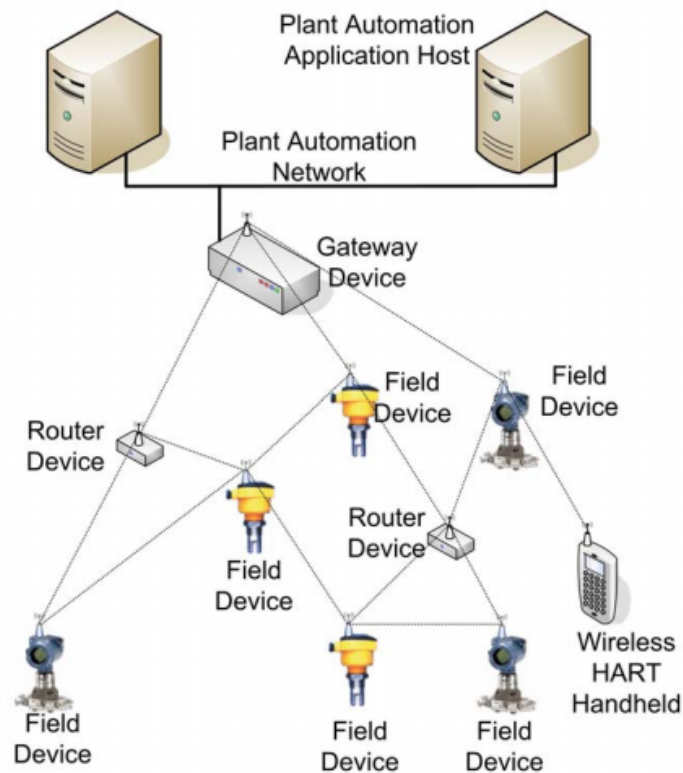


Figure 1.4: WirelessHART Mesh Networking [2].

On the other hand, Bluetooth 4.0 or Bluetooth Low Energy is a new protocol, designed over the classic protocol to overtake some drawbacks like power consumption and implementation [27]. This new approach presents the following characteristics:

- 150 meters of range (open field).
- 10mW (10dBm) output power.
- 15mA of maximum current.
- 1 μ A of sleep current.
- Built over star topology.

The main goal to be achieved with the development of this specific protocol was to enable its implementation on a coin cell based device. This protocol also presents an interesting advantage of allowing these small devices to communicate with mobile devices which proves to be a great asset for monitoring purposes.

Unlike the classical Bluetooth, which uses 79 1-MHz-wide channels, BLE uses 40 2-MHz wide channels. Three of those channels, are used for advertising and service discovery and are called advertising channels while the remaining 37 data channels are used to data transfer [4]. The advertising channels are used when the communication is being established and once that

occurs, the radio channel is changed to one of the data channels. At this point, the advertiser becomes a master and the listener becomes a slave, which waits for further instructions given by the master. This last specification allows the device to go into sleep mode, which offers this protocol the possibility to compete with others which provide low energy consumption as a key feature.

Last but not least, as mentioned in [28], SimpliciTI network protocol is a proprietary, low-power RF protocol targeting simple, small RF networks (less than 100 nodes). This protocol is designed for easy implementation with minimal microcontroller resource requirements. It is intended to support customer development of wireless end user devices in an environment in which, the network support is simple and the customer desires a simple way for transmitting data wirelessly. The protocol runs on *Texas Instruments* (TI) MSP430 ultra-low-power microcontrollers and is currently available for TI's CC1100/2420/2500 and other chips originating from those. The typical over-the-air data rate is 250 kbit/s, working in 480, 868, 915, 955 MHz and 2.4 GHz frequencies within a 100 meters range.

As shown in figure 1.5, SimpliciTI's architecture is divided in Application, Network and Data Link/PHY layers.

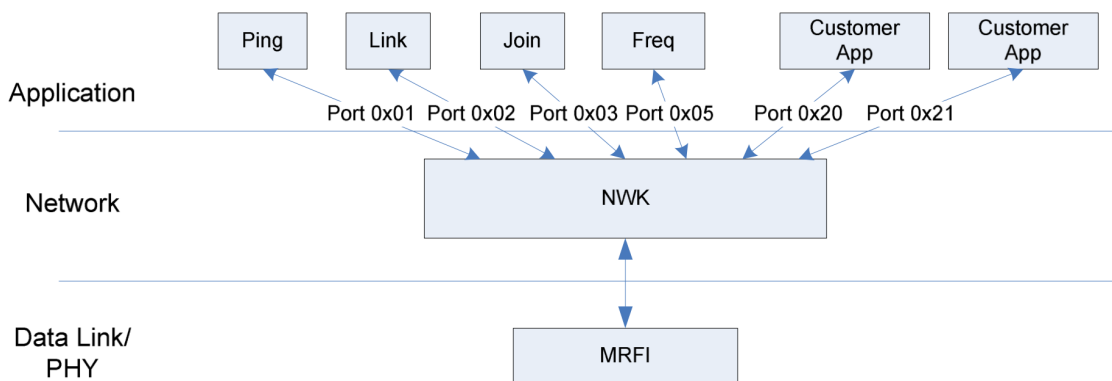


Figure 1.5: SimpliciTI protocol architecture [3].

The Application layer refers to the development section, and is comprised of a set of commands:

- Ping: used to detect the presence of a specific device.
- Link: enables the link management of two peers.
- Join: guards entry to the network in topologies with access points.
- Freq: used to change the communication frequency.

The Network layer is responsible for managing the received and transmitted queues. The last layer (Minimal RF Interface) does the actual writing and reading to and from the RF transceiver.

The protocol is oriented around application peer-to-peer messaging and in most cases the peers are linked together explicitly. However, SimpliciTI will support scenarios in which explicit

linking between pairs of devices is neither needed nor desired, therefore propagating the message throughout the network [3].

Table 1.3 provides a comparative analysis of the ZigBee, BLE and Simpliciti protocols. Since WirelessHART is based on a mesh network, which is not intended for this thesis, there is no point in comparing it with the rest of the protocols.

Table 1.3: Protocol comparison [4, 5].

	Bluetooth Low Energy	ZigBee	SimpliciTI
Advantages	<ul style="list-style-type: none"> - Low power. - Low cost. - 150m range. - Can communicate with mobile devices. 	<ul style="list-style-type: none"> - Low power. - Low cost. - 100m range. - Free source code. - Easy implementation. - Scalability. 	<ul style="list-style-type: none"> - Low power. - Low cost. - Range 30m. - Free source code. - Simple implementation. - Supports sleeping devices.
Disadvantages	<ul style="list-style-type: none"> - Restrictions regarding network topology. 	<ul style="list-style-type: none"> - Complex implement. - Requires 128k flash memory. 	<ul style="list-style-type: none"> - Restrictions regarding network topology.
Main features	<ul style="list-style-type: none"> - Star topology 	<ul style="list-style-type: none"> - Star, p2p or mesh topology. - Up to hundreds of nodes. - Code size: 50 to 60k. 	<ul style="list-style-type: none"> - Star or p2p topology. - Up to 30 nodes. - Code size: 4k. - Basic API.
Experimental Results	<ul style="list-style-type: none"> - Power consumption: 0.147mW. - Tx power range: 23 dBm. - Rx sensitivity: 90 dBm. - Sleep current: 0.5 μA. 	<ul style="list-style-type: none"> - Power consumption: 35.706mW. - Tx current: 18.5 mA. - Rx current: 5.4mA. - Idle current: 4mA. 	<ul style="list-style-type: none"> - Tx current: 23 mA. - Rx current: 19.4mA. - Idle current: 19 mA. - Tx power range: 55dBm. - Rx sensitivity: 87 dBm. - Sleep current: 0.9 μA.
Hardware Requirements	- Requires RF CC2540	- Requires RF CC2520	- Requires RF CC2500

WSN's find applications in the most diverse fields making possible to monitor and control, and operate on systems, environments, equipment, and habitats.

An example of extreme environment is targeted in the volcano monitoring system which is equipped with low-frequency acoustic sensors to monitor volcanic activity. This network has an event-detection mechanism to reduce the amount of data that needs to be communicated and processed. The combination of seismic and acoustic signals in a sensor array has great potential for assessing eruption intensity and interpreting trends in volcanic activity [29].

A WSN deployed over sensor-enhanced toys and other classroom objects supervises the learning process of children. The Smart Kindergarten project allows unobtrusive monitoring by the teacher [30].

The military field has also been an object of study and DARPA's self-healing minefield is proof of that. Anti-tank mines communicate over a peer-to-peer network forming a WSN capable of responding to attacks and redistributing mines in order to heal breaches in the field and to make it difficult for enemy troops to progress. Also, a wireless network using acoustic sensors has been tested in order to determine sniper localization. After a gun shot, the sound emitted is captured by the sensors to estimate the position of the sniper. This information is then forwarded to the BS.

The given examples show some of the fields that can be improved by the use of WSNs. Despite the few given, there are many other fields that can take advantage of the deployment of this type of technology.

1.2.3 Related Work

The evolution of parameters sensing, wireless communication and the combination of both, has increased the number of applications to monitor and control processes.

In recent times, the need to develop new ways of monitoring and controlling bioprocesses has been pointed out, and due to the developments referred above, it is now possible to implement a system of remote monitoring which is able to make local measurements and subsequently send the data to a central station so that the values can be observed and treated.

An example of an existing solution on the market is the 20 Series Wireless (mesh-based network) which is offered by Insite IG [31]. As shown in Figure 1.6, this system consists in sensors (DO and pH), a sensor transmitter and a process transmitter.

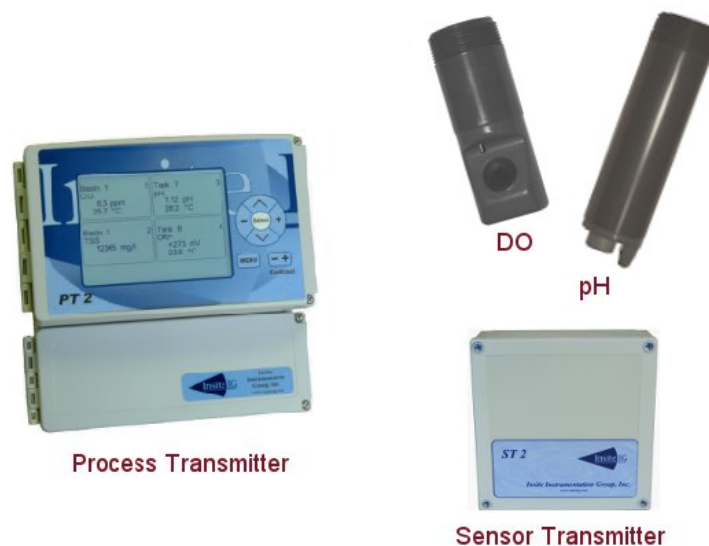


Figure 1.6: 20 Series Wireless by InsiteIG.

The process transmitter is the heart of the wireless mesh network able to contain up to sixteen sensor nodes containing pH and DO sensors. On one hand, to measure DO the system uses optical sensors and on the other hand, pH is measured with flat glass electrodes. All the sensor nodes are connected to a sensor transmitter. These supply power to the sensor and transmit data to the process transmitter.

According to what was reviewed, and to hardware and software requirements, there are some deductions to be made. First, due to the fact that the boards, which are described in chapter 2, use a CC2500 radio chip, the BLE and ZigBee protocols will not be implemented. Secondly, after reviewing the SimpliciTI protocol, it was established that this one is too complex and extensive for what is required in this application. These considerations lead to the conclusion that the best approach is to develop a new communication protocol.

1.3 Structure of the Thesis

In order to properly present the accomplished work, a brief preview is now provided for the following chapters, summarizing the content in each.

Chapter 2 gives a general overview of the designed system. It resumes the requirements of the system and it presents a structure breakdown of the elements that compose the integrated system. This chapter proves to be important since it introduces both hardware and software components, that will be discussed in detail throughout this document.

Chapter 3 will describe the wireless network protocol that was developed and implemented. First, the process of building the network is reviewed. It is followed by a detailed explanation of the communications that can occur when the network is being established and after that.

In Chapter 4, it is described the code that was written in order to fulfill the requirements previously given and to implement the wireless protocol that was presented in Chapter 3. The last discussed topic is related to the results of the tests that took place.

Finally, the Chapter 5 concludes the thesis with a discussion of the results and suggestions for future work.

Chapter 2

System Design

2.1 System Requirements

The purpose of designing and implementing the WSN system is to use it for measurement of various parameters (pH and DO) in chemical and biological processes. These processes are conducted under certain conditions, which in turn become system requirements for this thesis. They are as follows:

- The sensor board (SB) should be small enough to fit inside an incubator.
- The SB should work at 37°C, since this is the temperature at which most biological processes take place.
- Energy consumption must be optimized.
- The network should work with upto 16 nodes, so as to simultaneously monitor a number of parameters for a number of bio processes.

Note: Given that this is a prototype version, the network will be composed of only one BS and two SBs.

2.2 Structure Breakdown

Before introducing the protocol developed, it is important to distinguish and understand the elements that integrate the system where the work is focused. Here, the hardware used by this system and the developed software are also introduced.

This system is designed on 3 levels, which are shown in the diagram of the figure [2.1](#). These levels give a better understanding about how the system operates and behaves. The lower level, comprises of one or more wireless SBs. The middle level is the combination of the BS and a Personal Computer (PC), which corresponds to the data collection. The last and higher level is the Visual Interface (VI), which allows the end user to communicate with the devices in a simpler and ergonomic way.

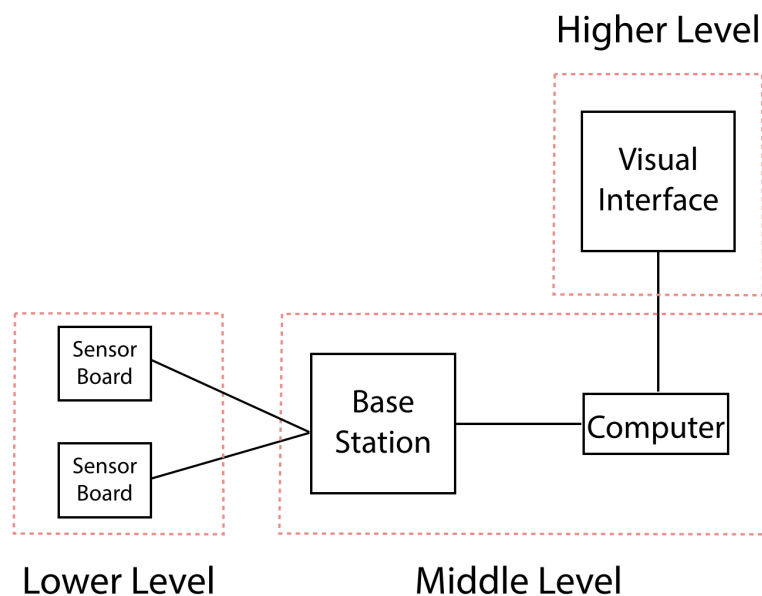


Figure 2.1: System architecture block diagram.

2.2.1 Lower Level: Sensor Board

In the lower level, each node consists of a SB, which is described below. The block diagram in figure 2.2 shows the components assembled on the SB.

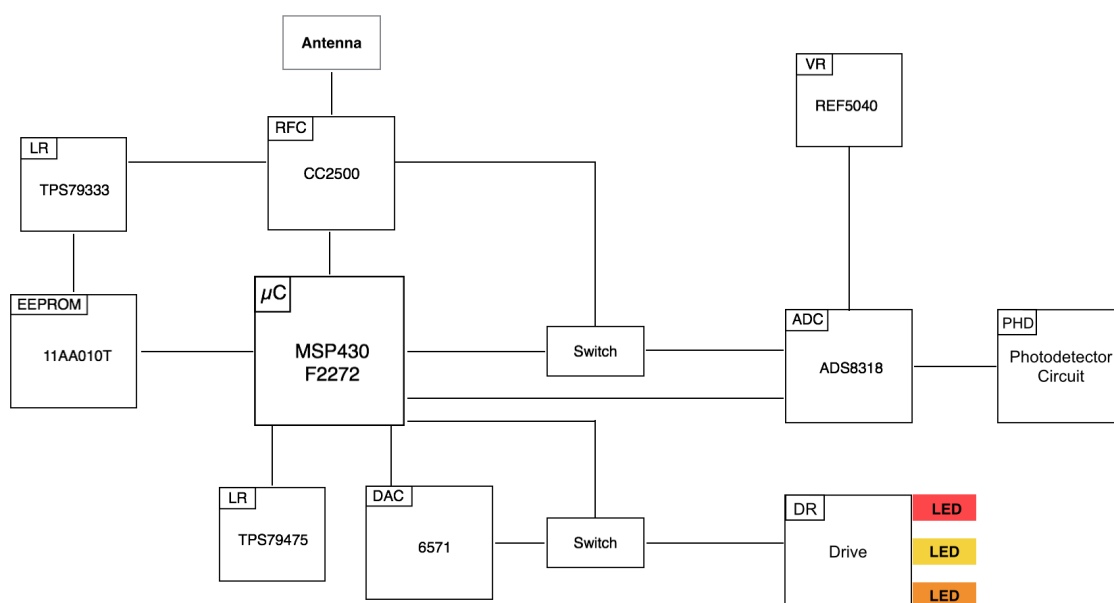


Figure 2.2: Sensor board block diagram.

The main component of this board is the MSP430F2272, which is a 16-bit ultra-low-power microcontroller, with 32Kb of Flash and 1Kb of RAM, and it is represented in the diagram as μC . With five low-power modes and features like low supply voltage (1.8 - 3.6V), ultra-fast wake

up from standby modes, Universal Serial Communication Interface (USCI), and ultra-low power consumption, this μ C proved to be an excellent solution for the system, since the application requires the battery life and the communication speed to be optimal [32]. It is also important to refer that the 32Kb of flash memory is divided in two, main memory and information memory. The information memory consists of four segments of 64 bytes each and it will be used when implementing some of the features of the system.

Another key component is the one that is responsible for the low-power wireless application. It appears in the diagram as RFC that corresponds to the CC2500 radio frequency chip, and together with the Antenna, it enables wireless communications which is the most important requirement for the success of this implementation. This chip was chosen because of characteristics like the output power up to +1dBm, the Serial Peripheral Interface (SPI) which is part of the USCIs supported by the μ C, and the low current consumption with programmable data rates up to 500 kBaud [6].

The component ADS8318, represented as ADC in the figure, is a 16-bit analog-to-digital-converter and it is connected to the μ C using a Serial Peripheral Interface (SPI) connection which is a synchronous serial data link based in a master/slave communication [33].

To be able to convert the digital data from the μ C to an analog signal, the selected component was a DAC6571, as shown in the diagram. This Digital-to-Analog Converter (DAC) with a 10-bit resolution is, like the others components, low-power and has very fast data transmissions, with speed up to 3.4 Mbps.

This board has the purpose of quantifying the pH and DO elements, and in order to do that, it is essential to collect data from the sensors which is acquired with the help of two circuits, first the photodetector circuit (PHD) and second the LED Drive (DR). Since this project is a prototype device, the PHD circuit did not take place. On the other hand, the work was fully focused on the DR. The reason why the DAC was placed between the μ C and the DR is because it is necessary to control the LEDs included in this last bit of circuit. As seen in figure 2.2, there are three LEDs, one red, one yellow, and one orange, which are very important to demonstrate the successful implementation of the wireless protocol developed.

The whole board is powered with a 6V battery. Figure 2.3 illustrates the SB used in the project.

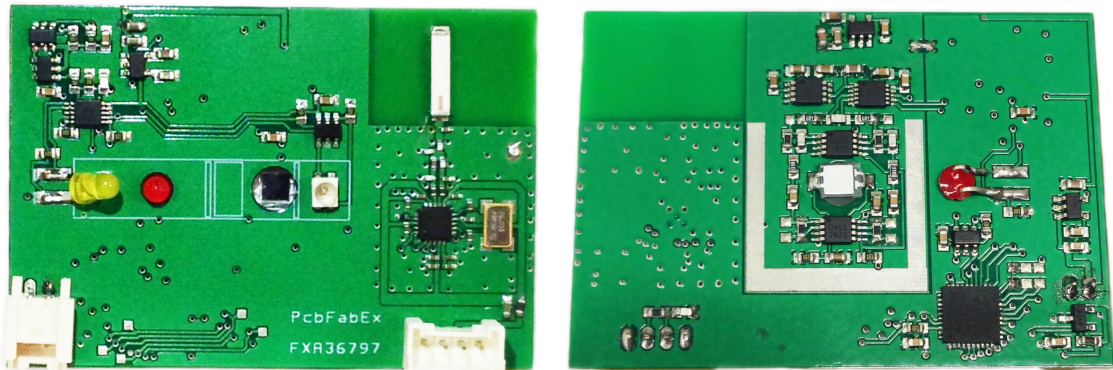


Figure 2.3: Sensor board top and bottom views.

2.2.2 Middle Level: Base Station and Personal Computer

As mentioned before, the middle level consists of two parts, an usual PC and a BS which is intended to be the master of the system. The diagram in figure 2.4 gives an overview of the elements that compose the BS.

Just like the SB, the main component of this board is the μC , but in this case, the one assembled was an MSP430F2471 [34]. Both μC s have similar features, with the RAM size - 4Kb in this case, being the major difference, which was required since the SB is supposed to collect information from 16 nodes.

To be able to communicate with the rest of the nodes, the BS also requires a wireless module composed by one CC2500 chip and one Antenna.

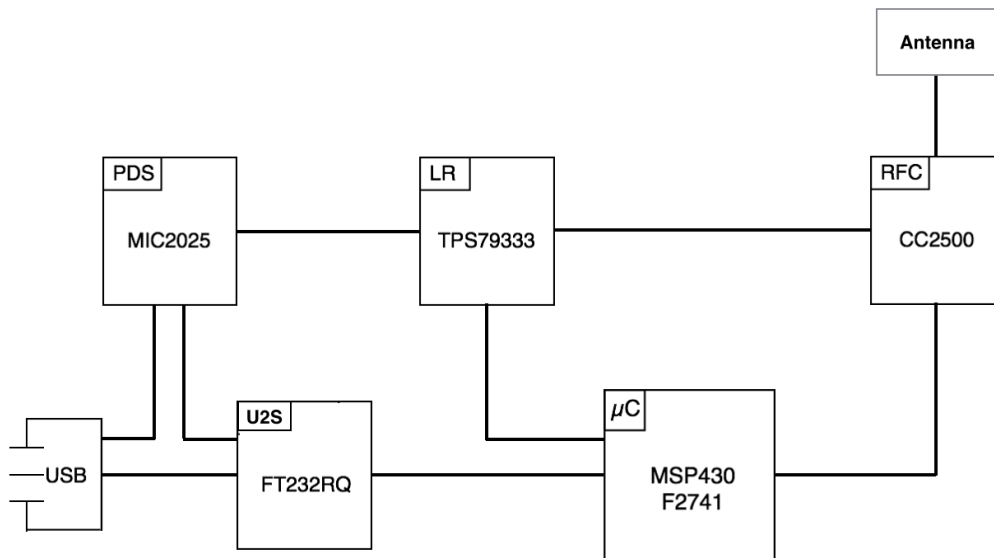


Figure 2.4: Base Station board block diagram.

The two way communication of the μC and the PC, which is wired, is achieved with the help of two components, an USB plug that connects to the PC on one side, and on the other side it is connected to an FT232RQ chip, mentioned as U2S in figure 2.4, which is an USB to serial Universal Asynchronous Receiver/Transmitter (UART) interface. This component also fulfills the requirement for speed, with high data rate, that goes up to 3 Mbps [35].

In order to download the code from the PC to the μC , the SB requires a 7-pin connector, which connects to a FET-Debugger used to accomplish the download.

As for the power supply, since the BS is connected to the PC through an USB port, this will be supplied with 5V.

The BS used in the project is illustrated in figure 2.5 and the schematic representation of the BS and SB are given in A.



Figure 2.5: Base station top and bottom views.

2.2.3 Higher Level: Visual Interface

In order to optimize the way that users interact with the network, it was necessary to create a VI, which was accomplished using *LabVIEW*. The VI enables the user to interact with the network by simply clicking one of the buttons of the menu and performing one of the commands. These will be explained in detail in chapter 3. Figure 2.6 is a visual illustration of the VI.

Select Device	Message Log	
Device 1 Device 2 All Devices	Waiting for command.....	
Select Command	Data Payload	Description
Red LED		This Command will turn ON the Red LED on the Sensor Board(s) during 3 seconds
Orange LED		This Command will turn ON the Orange LED on the Sensor Board(s) during 3 seconds
Yellow LED		This Command will turn ON the Yellow LED on the Sensor Board(s) during 3 seconds
Write Data	Type Data: 0 0 0 0	This Command will send data to be written on the Sensor Board's flash memory
Read Data	Out data: 0 0 0 0	This Command will return the data read from the Sensor Board's flash memory
STOP		

Figure 2.6: *LabVIEW* visual interface.

The menu is divided in five sections, one for the commands - "Select Command", one for the data to be sent (if applied) - "Data Payload", and one with a brief description of the command - "Description". The fourth section is responsible for the selection of the devices that will be forwarded a command - "Select Device". The last section - "Message Log" gives information about the status of the process.

The first section gathers a set of commands, that can perform some temporary or permanent changes in one or more nodes. There are five buttons in the menu, each one for one command. The first three buttons are responsible for interacting with the LEDs from the DR relative to each node, while the last two buttons enable the writing and the reading from and to the flash memory of their μ Cs.

The second section just takes place when the command *Write* or *Read Data* are intended to be used. In this section there are four input (or output for the command *Read Data*) fields (1 byte for each), which is the data to be sent from the BS to the node(s) which must be filled by the user.

Section three gives information about what is expected to happen when the button is pressed. The fourth section helps the user to properly address a command selecting the desired device.

Also, the user can send a command to device 1, device 2, or both, which means, broadcasting the message, which is enabled by the fourth section.

Finally, the last section helps the user understanding what is the state of the process of data transmission. For example, a message - "Command was successfully sent to sensor board" is shown if the process is complete.

Chapter 3

Wireless Network Protocol

This chapter presents the structure of wireless protocol that was developed, which is divided in four sections. The first section introduces the setup of the network. Section two explains the communication from the PC and the BS. The third section describes the two way communication between the BS and the SBs while the fourth section is responsible for specifying the interaction of the BS with the PC. Given that the protocol was optimized to work with one BS and two SBs, the network behavior will be explained under these conditions.

3.1 Network Setup

First of all it is important to understand that each board, whether BS or SB, was given a Serial Number, consisting of one letter and one number. For example, S1, which means, Sensor number 1, and B1 which means, Base number 1. These will help both the BS and the SB addressing messages and checking which board sent or received messages.

The network starts when the BS is plugged in the PC. This will wake up μC and run a set of instructions, mainly configurations. These will be discussed in the next chapter. Once the BS is operational, it will wait for a SB to join the network.

The next step is to turn on one of the SBs activating the battery. After starting up, the SB's μC will also run some configurations in order to enable the communication with the BS. After that, a message will be sent automatically to the BS requesting permission to join the network. This request is sent via Antenna and it is organized in three groups of data that together build a frame. Figure 3.1 presents the request frame composition.



Figure 3.1: Request frame.

The first group, with size of 1 byte, corresponds to the length of the rest of the frame. For example, a frame of 7 bytes should have a length of 6. With a 2 bytes size, the second group holds information about the source address which in this case will be "S1", 1 byte occupied with the letter "S" and the other byte occupied with the number "1". The final group contains a 4 bytes password representing a sequence code to get inside the network. In this case the password is the word "UMBC", and each byte is occupied with each letter. Figure 3.2 gives the request frame sent by the SB1.



Figure 3.2: Request frame example.

After the BS receives the request, it confirms if the password is correct. If that is true, it saves the address of the device that sent the request, for future purposes. Now it is time to grant the device access to the network and in order for this action to take place, a new type of frame will be used. As shown in figure 3.3, this frame consists of seven groups.

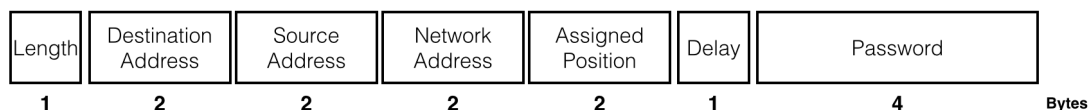


Figure 3.3: Grant permission frame.

Similarly to the previous, this frame also contains, the length of the following data, the source address which will be "B1", and the password which is also different this time - "FEUP". There are four new groups in this frame. Destination address holds, in 2 bytes, the address of the SB - "S1". Using 2 other bytes it is also given information about the network address, which was defined as "N6" (Network 6). Another important information contained in the frame is the position that was assigned to the SB when communicating within the network. With 2 bytes, the assigned position is "P1" (Position 1). The last information given has 1 byte length and carries the delay in seconds. This delay (for future purposes) represents the time that the SB needs to wait before running commands. This topic will be reviewed in detail further. As for now, since this is the first SB joining the network, the delay will be "0".

An example of the presented frame is given in figure 3.4.

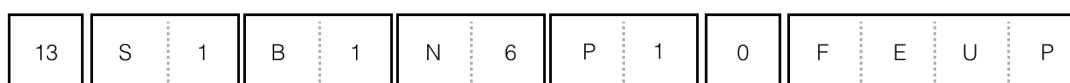


Figure 3.4: Grant-permission frame example.

When this frame is captured by the SB, it needs to be properly read. Again, the first thing to be checked is the password. After the password is confirmed, it is also necessary that the destination address matches with the one that the SB holds. Once this is also confirmed, the μC saves the information of the network address, the BS's address, the assigned position, and the delay. Finally, in order to visually inform the user that the communication was properly established, the red LED on the SB will blink three times.

It is noteworthy that at this point, the communication was established and the network, although not completely populated, is fully operational with the SB that joined the network. In order to fill the network, the second SB is turned on and the whole process described above takes place one more time. Figure 3.5 presents a state machine from the network's point of view. It provides information about the behavior of each board while the network is being populated.

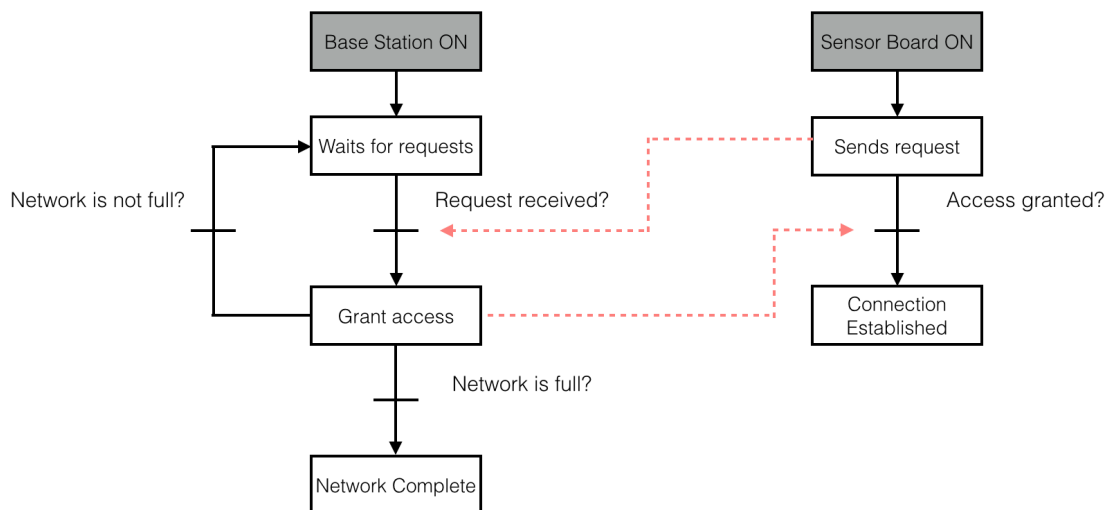


Figure 3.5: Network's state machine.

It should be noted that the BS continues listening for a request to join the network from the second SB, while simultaneously waiting from instructions sent from the PC. This will be explained further in the following section.

3.2 Communication from the Computer to the Base Station

In this section it is assumed that there is only one SB in the network. At this point, the network is on waiting mode. This means that the SB is waiting for instructions from the BS. On the other hand, the BS is waiting for the PC to communicate with it, which happens when the user interacts with the VI.

As mentioned before, there are five commands that can be executed by the SB. In order to tell the SB what to do, the user must select one of the commands existing in the menu of the VI. Table 3.1 list the commands that can be processed by the SB as well as their description.

Table 3.1: List of commands and its descriptions.

Command	Description
1	Red LED turned ON during 3 seconds
2	Yellow LED turned ON during 3 seconds
3	Orange LED turned ON during 3 seconds
4	Writes 4 bytes data on the flash memory of the SB
5	Reads 4 bytes data from the flash memory of the SB

Since each one of these commands is associated with one button, when the user presses one of the buttons, an instruction is sent to the BS. This instruction is a group of, either 2 bytes or 6 bytes, depending on the command. For the commands 1, 2, 3, and 5, the size of the instruction is 2 bytes and for the command 4, the size is 6 bytes.

The first 2 bytes of each instruction, carry, first the number of the command, and second the destination number, which is the number of the network position of the SB that is going to operate. The destination can be "1" to communicate with the first device that joined the network, "2" to communicate with the second device, and "FF" to broadcast the instruction to both SBs. The broadcast mode will be explained in the section 3.5 of the present chapter. As an example, if the user wants to communicate with the SB, and the command to be executed is "2", the instruction sent will be as shown in figure 3.6 .



Figure 3.6: Command 2 Instruction frame.

As for command 4 instruction frame, the last 4 bytes will contain the data to be sent. This data is specified by the user in the VI. The example given in figure 3.7, shows the instruction frame for command 4 when the user send the data "WXYZ".



Figure 3.7: Command 4 Instruction frame.

After the BS receives the instruction it is time to properly build a command frame to be sent to the SB. This frame is 40 bytes long. The generic form is shown in figure 3.8.

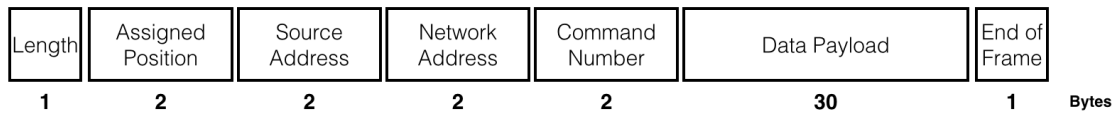


Figure 3.8: Command frame.

There are seven groups that compose the frame. Some of these are the same as the ones introduced before, namely, the length, the assigned position, the source address, and the network address. The other three groups are named command number, the data payload and the end of frame.

With 2 bytes size, the command number is the combination of one letter "C" and one number, corresponding to the number of the command that the user wants to be executed. By default, the data payload, which is a group of 30 bytes, is filled with zeros. Except when requesting for command 4, this group retains default value. But when it comes to command 4, the first 4 bytes take the value of the data inserted by the user. Finally, the last group, represented by 1 byte, indicates whether the frame reached to the end or not. If the last byte has the value "1", it means that there is no more data to receive. On the other hand, if the value is "2", it means that the SB will receive at least one more frame of 40 bytes. This specific group is intended for future purposes, for transmissions of 100-150 bytes. The figure 3.9 gives an example of the frame when requesting the command 4.

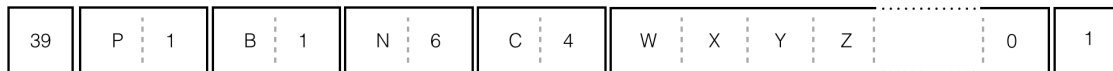


Figure 3.9: Command 4 frame example.

3.3 Two Way Communication Between Base Station and Sensor Board

After receiving the command frame, the SB needs to process the whole data in order to interpret the action to be executed.

First, the destination, source, and network addresses are confirmed. Once this step is over and validated, information about the command is saved. Recalling that the network has only one SB, delay will not take place before executing the commands.

3.3.1 Processing Commands 1, 2, and 3

These three commands, which are linked to the three LEDs, have the same work flow. Commands 1, 2, and 3 are associated to the red, yellow and orange LEDs, respectively.

When the command is acknowledged, the μ C is responsible for triggering the corresponding LED. It will remain active for 3 seconds, and then will be turned off.

At this moment, both the SB and BS will be waiting for new instructions.

3.3.2 Processing Command 4

This command is responsible for manipulating the flash memory of the μ C of the SB. After acknowledging the command, the next step is to fetch the data inserted by the user. As soon as the data is acquired, it is written in the flash memory.

Although not specified to work directly with the LEDs, this command uses two of them, the red and the orange, in order to provide a visual aid to the user. Therefore, when the command is executed, these two LEDs will simultaneously be turned on for 1/2 second, indicating that the process of writing was successfully performed.

It should be noted that every time this command is executed, the section of the flash memory that is used, is always the same. Thus, this is a process of overwriting data.

3.3.3 Processing Command 5

The process of reading the existing content in the flash memory is accomplished by command 5. It is very similar to the first three commands, except for the fact that this command demands an answer from the SB, after execution. This requires that the communication also flows in the opposite direction, in other words, from the SB to the BS.

Therefore, after reading the data, it is necessary to build a new frame to be sent to BS. This frame is the same as the one illustrated in figure 3.8. Once the frame is complete, it is sent to the BS via Antenna.

Again, in this command the LEDs are used to help the user understand if the reading process was executed properly. In case everything occurs as intended, the red and the orange LEDs will blink twice in periods of 1/8 of a second.

3.4 Communication from the Base Station to the Computer

As mentioned before, this flow of communication only takes place when command 5 is executed. So, the moment that the BS sends the command frame it is known that an answer is going to be received soon.

The way the response from the SB is processed is as simple as reading the content from the frame group Data Payload, which will be a total of 4 bytes. As soon as the reading is complete, the data is sent through the USB to the PC. This way, the user is able to visualize the data in the VI.

3.5 Network Operation with Two Sensor Boards

The network is considered to be complete when there is one BS and two SBs. The operation with the network fully populated is similar to the one described before, except for a few changes.

As mentioned in section 3.1, after the first SB is operational, the BS is set to waiting mode, listening either for instructions from the PC or for new requests to join the network. At this

point, the second SB is turned on, and with permission given by the BS, joins the network, with the corresponding delay of operation. From now on, the user can communicate with the first or second SB, or with both at the same time. It should be noted that the all of the frames presented before, remain the same when it comes to size and group formation. The only thing that changes is the address of the SB. This means that, when the user sends a command to the first SB, the address is "1", while the second SB has the address "2". If the intent is to broadcast the message, the address is "FF".

When the message is addressed to the second SB, the process is exactly the same as when communicating with the first one.

Assuming the message is broadcasted, it is important to understand the behavior of the SBs when these are requested. The next figure illustrates an example of a broadcasted frame requesting command 1.

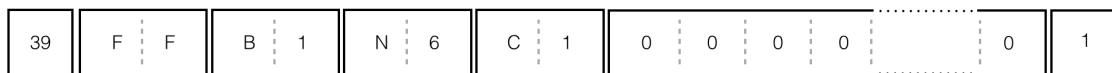


Figure 3.10: Command 1 frame for broadcasting.

For commands 1, 2, and 3, the only thing that changes is the time of operation. This means that, after both boards have acknowledged the command, the first SB operates immediately and the second board operates after the delay that was imposed by the network, which in this case is 1 second. Resuming, taken the example from the figure 3.10, the red LED of the second SB will be activated 1 second later than the one from the first board.

As for command 4, the delay is not taken into account. Therefore, both boards execute the command simultaneously.

Command 5 was not implemented for broadcast and it is considered to be future work.

Chapter 4

Results

This chapter, divided in two sections, presents the methods used to successfully meet the system requirements. The first section will describe the code that was implemented. In the second section are the results of the tests carried out after the completion of the implementation.

4.1 Code Description

4.1.1 Base Station Code

On a first note, the code was developed using *C programming language*. This task was performed using the software *IAR Embedded Workbench*. It is also important to understand that the entire code implements a star topology network.

The first thing to do, as mentioned before, is connect the BS to the PC. Once the device is supplied with power, the μ C starts and runs the program previously downloaded.

Figure 4.1 presents the *Main()* function, which is where the program starts its execution. It is a simple function with only two objectives. First, it runs global configurations with help of another function, *Setup_Init()*. These will configure the frequency clock, the ports and pins that will be used, and the RFC will be set for receiving mode. After that, it puts the μ C in sleep mode (global interruptions enabled), which is achieved with the low-power mode 4, in order to maximize battery life. This way the BS will only operate again when an interruption occurs.

```
// MAIN FUNCTION
void main()
{
    Setup_Init();
    __bis_SR_register(LPM4_bits + GIE); // sleep until next interrupt service routine (low power mode 4)
}
```

Figure 4.1: Function *Main()* implemented in the Base Station.

For this system, the interruptions can occur either when the radio is triggered after a signal is received, or when a message is received from the PC. These events serve the purpose of waking up the μ C, however it is necessary to define the set of instructions that will be invoked to take care

of the condition that caused the interrupt. Note that each set of instructions comprises an Interrupt Service Routine (ISR) and each interruption has its own ISR.

At this point, the BS is the only device existing in the network. The process of populating the network will only be executed once this device receives one or more requests from the SBs. This will trigger one of the two ISRs, which is responsible for managing the interrupts based on radio events.

<pre> // INTERRUPT SERVICE ROUTINE (ISR) RESPONSIBLE FOR ACQUIRING DATA SENT FROM THE SENSOR BOARD(S) // (REQUEST TO JOIN THE NETWORK OR DATA SENT DUE TO EXECUTION OF THE COMMAND 5) #pragma vector=PORT2_VECTOR __interrupt void Port2_ISR(void) { if(TI_CC_GD00_PxIFG & TI_CC_GD00_PIN) { uint8_t status[2]; // Buffer to store frame status bytes static uint8_t crcOk; // flag variable to check if the received frame is good if(devices_connected>=0 && devices_connected<=2) { </pre>	
1.	<pre> uint8_t len = 0x06; // lenght of the frame that is intended to receive = 6 bytes uint8_t rx[6]; // buffer to store received frame uint8_t rx_2[39]; crcOk = RFReceivePacket(rx,rx_2,&len,status); // fetch packet from cc2500 </pre>
2.	<pre> if(crcOk==0x80) // condition to manage network { *** } </pre>
3.	<pre> else if(crcOk==0x50) // condition to acquire data payload due to command 5 request { *** } </pre>
<pre> } } TI_CC_GD00_PxIFG &= ~TI_CC_GD00_PIN; // resets flag if(devices_connected<2) { TI_CC_SPIStrobe(TI_CCxx0_SRX); // sets cc2500 to RX mode } } </pre>	

Figure 4.2: Interrupt Service Routine for radio events implemented on the Base Station.

As said before, this ISR will be executed after a radio event occurs, which in this case means that it will be executed after a request to join the network or after receiving data payload due to execution of command 5. As confirmed in figure 4.2, the flow of this routine consists of the following steps:

1. Collecting data via radio, using function *RFReceivePacket()*.
2. If the collected data is related to network management, executes a snippet of code (Please see details in [B.1.1.1](#)).
3. If the collected data is related to command 5 request, a different snippet of code is executed.

Step 1 is accomplished using the function in figure 4.3. This function will fetch the received frame. It will update the *rx_Buffer* in case a SB wants to join the network or it will update *rx_Buffer_2* in case SB is already communicating and answering with data. Therefore, the *rx_Buffer* will contain the request frame, and the *rx_Buffer_2* will contain the command frame.

For each case of update this function will return a different value, in order to define the next set of instructions to be executed. It will either execute step 2 or 3 from the previous list. Step 3 will be explained in detail later.

```

char RFReceivePacket(char *rxBuffer, char *rxBuffer_2, char *length, char *status)
{
    char pktLen;
    if ((TI_CC_SPIReadStatus(TI_CCxxx0_RXBYTES) & TI_CCxxx0_NUM_RXBYTES))
    {
        pktLen = TI_CC_SPIReadReg(TI_CCxxx0_RXFIFO); // Read length byte
        if (pktLen <= *length)                       // If pktLen <= rxBuffer
        {                                             // Then save packet
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, rxBuffer, pktLen); // Get data
            *length = pktLen;                                     // Get Len
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, status, 2);    // Get stat
            return (char)(status[TI_CCxxx0_LQI_RX]&TI_CCxxx0_CRC_OK); // Ret OK
        }
        else if(pktLen==0x27)
        {
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, rxBuffer_2, pktLen); // Get data
            *length = pktLen;                                     // Get Len
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, status, 2);    // Get stat
            return 0x50; // Ret OK
        }
        else
        {
            *length = pktLen;                                     // Return the large size
            TI_CC_SPIStrobe(TI_CCxxx0_SFRX);                     // Flush RXFIFO
            return 0;                                             // Retn Error flag
        }
    }
    else
    {
        return 0; // Retn Error flag
    }
}

```

Figure 4.3: Function *RFReceivePacket()*.

Since the network is not populated yet, the first thing that will occur is an interrupt due to request from the first SB. This code will operate according to the information acquired previously.

First, the password "UMBC" will be confirmed. If true, the address of the SB is saved. Finally, the BS sends the permission using the function *Send_Network_Permission()*, shown in figure 4.4, which builds the grant-permission frame and transmits it to the matching SB. Since the *RFSendPacket()* function is very similar to the one implemented for receiving frames, this will not be described. However, it can be found in section B.1.1.2 of B.

In case there is a second SB requesting for permission, the process will be repeated. After this, the μC and the RFC are set to sleep mode. Once the network is populated (either with one or two SB) it is possible to interface the BS and the PC. This action takes place when an interrupt occurs due to events based on UART communication. Hence, proving to be a reason to implement a different ISR (Please see details in B.1.1.1).

As mentioned before, the interaction of the PC with the BS, when requesting commands, uses instructions with 2 or 6 bytes depending on the command. The sequence of this instructions is: command number, destination address, and data payload, which can exist or not. The present code was designed to acquire these instructions in that specific order. After saving the 2 first bytes, the code will determine if its necessary to continue listening or not, based on the command. For command 4, the routine maintains the reading process and gathers the last 4 bytes.


```

// FUNCTION THAT SENDS THE PERMISSION TO JOIN THE NETWORK
char Send_Network_Permission(void) // formation of the grant-permission frame
{
    static char grant_access[14] = {0x00,0x00,0x00,0x42,0x31,0x4E,0x36,0x50,0x00,0x00,0x46,0x45,0x55,0x50};

    if(devices_connected==0)
    {
        grant_access[1] = sensor_addr_1[0]; // updates destination address using global variable
        grant_access[2] = sensor_addr_1[1];
        grant_access[8] = 0x31; // position assigned to the device
    }
    if(devices_connected==1)
    {
        grant_access[1] = sensor_addr_2[0]; // updates destination address using global variable
        grant_access[2] = sensor_addr_2[1];
        grant_access[8] = 0x32; // position assigned to the device
        grant_access[9] = 0x01; // delay imposed by the network (seconds)
    }
    uint8_t pkt_lenght = sizeof(grant_access); // frame size
    RFSendPacket(grant_access, pkt_lenght); // Send request to join the network

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // set cc2500 to Idle mode

    return 0x80;
}

```

Figure 4.4: Function *Send_Network_Permission()*.

Whatever the case may be, once the data collection is complete, the next step is building the frame, followed by a radio transmission. The function *Send_Command()*, given in figure 4.5, is responsible for this implementation and also for some validation of information. In other words, before the radio transmission, this function checks if the command and the address are listed. There is no point in sending the frame if one of these groups is invalid.

```

// FUNCTION THAT SENDS THE COMMAND TO BE EXECUTED BY THE SENSOR BOARD(S)
char Send_Command(char rxd_cmd , char rxd_dstn, volatile unsigned char *data_to_send)
{
    static char command_packet[40] = {0x27,0x00,0x00,0x42,0x31,0x4E,0x36,0x43,0x00}; // formation of the command frame
    if(rxd_dstn==0xFF)
    {
        command_packet[1] = 0x46; // destination addr is set to "FF" (broadcast message)
        command_packet[2] = 0x46;
    }
    if(rxd_dstn==1 || rxd_dstn==2)
    {
        command_packet[1] = 0x50; // destination addr is set to 'PX' with X being the device number
        command_packet[2] = 0x30 + rxd_dstn;
    }

    if(rxd_dstn!= 0x01 && rxd_dstn!= 0x02 && rxd_dstn!= 0xFF) // Check if destination is listed
    {
        return 0x10; // Incorrect Destination
    }

    if(rxd_cmd!= 0x01 && rxd_cmd!= 0x02 && rxd_cmd!= 0x03 && rxd_cmd!= 0x04 && rxd_cmd!= 0x05) // Check if command is listed
    {
        return 0x20; // Incorrect Command
    }

    command_packet[8] = 0x30+rxd_cmd;
    command_packet[9] = data_to_send[0];
    command_packet[10] = data_to_send[1];
    command_packet[11] = data_to_send[2];
    command_packet[12] = data_to_send[3];
    command_packet[39] = 0x31; // Set END OF PACKET to 1 (0x31) If 1, packet is over, if 2 packet is not over.

    uint8_t pkt_lenght = sizeof(command_packet); // frame size
    RFSendPacket(command_packet, pkt_lenght); // sends command to sensor board(s)
    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // set cc2500 to Idle mode

    if(rxd_cmd==0x05)
    {
        return 0x50; // successfull command 5 transmission
    }
    else
    {
        return 0x30; // successfull command 1,2,3 or 4 transmission
    }
}

```

Figure 4.5: Function *Send_Command()*.

After a successful transmission, there are two paths to follow. If the transmission contained one of the first 4 commands, the μ C will end the execution of this ISR and will return to sleep mode. If the command 5 was sent, the path to follow is a little different. The RFC needs to be immediately switched to receiving mode.

Here, the step 3 will be able to occur. A message received after the execution of the command will trigger once again the ISR relative to radio events. This message contains the data payload that was requested which requires special attention. After being properly acquired, the data needs to be shown in the PC, which is accomplished with the cycle in figure 4.6.

```

while(count<4) // acquires data payload
{
    while (!(IFG2&UCA0TXIFG));
    UCA0TXBUF = rx_2[6+count]; // sends data payload to the computer for further display
    count++;
}

```

Figure 4.6: Cycle that uploads the data payload to the Personal Computer.

This concludes the behavior of the BS when submitted to any kind of events.

4.1.2 Sensor Board Code

From this point of view, the network is still unpopulated but assuming the BS is already connected to device and properly configured, the SB(s) can now be started.

The program begins in a similar way to the one which was download into the BS. It also has a *Main()* function but with some slight modifications. The function responsible for the initial setup is somehow different, since, for example, there is no need to enable UART communication. On the other hand, it is necessary to configure a some of the pins of the μ C to allow the interface with the LEDs. Also, before going into sleep mode, the program executes one more instruction. The function *Join_Network()* from figure 4.7 builds the request frame into the variable *join_nw* and sends it using the same function used by the BS for radio transmission, *RFSendPacket()*. This function will write the frame in the registers of the RFC and will activate the transmission mode of the same device.

```
// FUNCTION THAT SENDS THE REQUEST TO THE BASE STATION TO JOIN THE NETWORK
void Join_Network(void)
{
    static char join_nw[7] = {0x06,0x53,0x32,0x55,0x4D,0x42,0x43}; // formation of the request frame
    uint8_t pkt_lenght = sizeof(join_nw); // frame size
    RFSendPacket(join_nw, pkt_lenght); // sends request to join the network
    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // set cc2500 to idle mode
    TI_CC_SPIStrobe(TI_CCxxx0_SRX); // set CC2500 to receive mode (waiting for answer)
}
```

Figure 4.7: Function *Join_Network()*.

It is important to refer that after the request has been sent, not only the μ C will be put into sleep mode, but the RFC will be set for receiving mode.

The code responsible for handling the interrupts triggered by the radio events was implemented the same way as in the BS.

```
// INTERRUPT SERVICE ROUTINE (ISR) RESPONSIBLE FOR ACQUIRING DATA SENT FROM THE BASE STATION
#pragma vector=PORT2_VECTOR
__interrupt void PktRxdISR(void)
{
    if(TI_CC_GD00_PxIFG & TI_CC_GD00_PIN)
    {
        uint8_t status[2];           // buffer to store frame status bytes
        static uint8_t crcOk;        // flag variable to check if the received frame is good

        1. if(device_status==0)      // check if device is not connected to the Network
            { ... }

        2. if(device_status==1)      // check if device is connected to the Network
            { ... }

        TI_CC_GD00_PxIFG &= ~TI_CC_GD00_PIN; // seset IRQ flag
        TI_CC_SPISStrobe(TI_CCxxx0_IDLE);    // set cc2500 to idle mode.
        TI_CC_SPISStrobe(TI_CCxxx0_RX);      // set cc2500 to RX mode.
    }
}
```

Figure 4.8: Interrupt Service Routine for radio events implemented on the Sensor Board.

As shown in figure 4.8, there are two possible ways of taking care of the interrupt. The first case takes place if the SB is not connected to network, and the second, if the SB is already connected. In other words, it will either join the network or execute a command. This is a way to determine the frame that is supposed to be fetched, once the grant-permission and the command frames differ in size.

Receiving the confirmation to enter the network implies that some code instructions are executed. It is necessary to check and save some of the information contained in the frame. In case the password is confirmed as "FEUP" and the destination address matches the SB's address, the next step is to save the rest of the information. In order to be able to operate under BS's solicitation, the program is responsible for saving four pieces of the frame. These are, the address of the BS, the address of the network, the assigned position given by the BS, and the delay of operation. The moment this task is completed, the status of the device is updated to "Connected" and the red LED of the SB will blink three times, indicating that the communication was established.

On the other hand, if the device has already established communication, this ISR will run a different set of instructions. An important information that needs to be the first to be validated is the destination of the command frame. It was mentioned previously that if the address is "FF", the SB will proceed in a specific way, but if the address corresponds exactly to its serial number, the execution occurs immediately.

Since the only thing that differs from case to case is the time that the operation is executed, the delay will be ignored for a better comprehension of the program flow.

Figure 4.9 presents the code responsible for activating the red LED, which is command 1. Commands for blinking the yellow and the orange LEDs are similar to this one. Therefore, they will not be described.

```
if(command[1]==0x31) // if the command to be executed is the command 1
{
    Set_RedLED();
    __delay_cycles(24000000); // 3 second delay
    Reset_LEDs();
}
```

```
void Set_RedLED(void)
{
    P4OUT |= 0x01; // 4.0 set to high! LED
    P4OUT &= ~0x40; // 4.6 set to low!
    P4OUT |= 0x20; // 4.5 set to high!
}
```

Figure 4.9: Functions implemented to enable the red LED.

As for the command that requires that some data is saved in the flash memory, the corresponding code is divided in two functions, *Setup_Flash_Clock()* and *Write_Flash_SegC()*. The process of writing in the flash memory itself, involves a group of procedures that are stated below [36].

1. Setting the frequency of the communication with the flash memory.
2. A Flash Key needs to be provided in order to unlock the registers.
3. Before writing, the flash memory must be erased.
4. Configure the μC to write mode.
5. Loop responsible for writing the data in segment C of the memory.
6. Locking the registers when writing process is over.

Figure 4.10 contains the instructions that implement those steps.

```

// FLAHS MEMORY CLOCK SETUP FUNCTION
void Setup_Flash_Clock(void)
{
    DCOCTL = 0; // select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_1MHZ; // set DCO to 1MHz
    DCOCTL = CALDCO_1MHZ;
    FCTL2 = FWKEY + FSSEL0 + FN1; // MCLK/3 for Flash Timing Generator
}

// FLAHS MEMORY WRITE FUNCTION
char Write_Flash_SegC(void)
{
    char *Flash_ptr; // flash pointer
    unsigned int i = 0;

    Flash_ptr = (char *)0x1040; // initialize flash pointer to segment C
    FCTL3 = FWKEY; // clear Lock bit
    FCTL1 = FWKEY + ERASE; // set Erase bit
    *Flash_ptr = 0; // dummy write to erase flash seg
    FCTL1 = FWKEY + WRT; // set WRT bit for write operation

    while(i<4)
    {
        *Flash_ptr++ = data_payload_rxd[i]; // write value to flash
        i++;
    }

    FCTL1 = FWKEY; // clear WRT bit
    FCTL3 = FWKEY + LOCK; // set LOCK bit

    return 0x10;
}

```

Figure 4.10: Functions *Setup_Flash_Clock()* and *Write_Flash_SegC()*.

The flashing of the orange and red LEDs will indicate a successful execution of the command.

For command 5, which is the process of reading the data from the flash memory, the operation consists of two parts. The first will perform the proper reading and the second will send it back to the BS.

Reading the data from the flash memory is easier than the writing into it. The only specification needed, is the address of the segment that will be read. The code related to that can be consulted in [B.1.2.1](#).

As soon as the data has been read and placed properly in a variable, again it is necessary to build a frame and proceed to the radio transmission. Function *Send_Data()*, in figure 4.11, helps the SB execute the process stated.

```

// FUNCTION THAT SENDS THE DATA READ FROM THE FLASH MEMORY TO THE BASE STATION
char Send_Data(void)
{
    static char data_packet[40] = {0x27,0x00,0x00,0x00,0x00,0x00,0x4E,0x00}; // formation of the frame to send data

    data_packet[1] = base_station_addr[0]; // updates destination address using global variable
    data_packet[2] = base_station_addr[1];
    data_packet[3] = position_number[0]; // updates source address using global variable
    data_packet[4] = position_number[1];
    data_packet[6] = nw_addr[1]; // updates network address using global variable
    data_packet[7] = data_payload_txd[0]; // updates data read from flash using global variable
    data_packet[8] = data_payload_txd[1];
    data_packet[9] = data_payload_txd[2];
    data_packet[10] = data_payload_txd[3];
    data_packet[39] = 0x31; // updates end of frame byte

    uint8_t pkt_length = sizeof(data_packet); // frame size
    RFSendPacket(data_packet, pkt_length); // sends data to base station

    return 0x50;
}

```

Figure 4.11: Functions *Send_Data()*.

4.2 Tests

After being fully implemented, the network was submitted to a set of tests. These tests are important because they indicate the extent to which the system requirements have been met. The parameters that were evaluated with the completion of these tests were, the efficiency of the hardware and functionalities of the VI.

4.2.1 Efficiency

It was necessary to define a set of conditions to properly conduct the efficiency tests which the system underwent. These conditions correspond to paths of communication, with or without obstacles, and to output power of the RFC. Their results are presented on a quantitative form that correspond to the range of the communications.

Three different scenarios were used to test the communication paths. These scenarios are intended to simulate the circumstances that this system may undergo in the future. The first scenario is the most direct of them since it is an obstacle-free test. The second consists of placing a SB inside a walk-in freezer and closing the door, while the BS is connected to a PC outside the freezer. It should be noted that this door is made of metal and it is 5 cm thick. The last scenario takes place inside a sealed incubator. It is operated like the previous one, one SB inside the incubator and the BS outside. It is important to mention that the incubator has two doors. One is made of glass (labelled (a) in figure 4.12), that is 0.5 cm thick and the other is made of metal (labelled (b) in figure 4.12), which has 5 cm of thickness. Between these doors there is a gap of 2 cm.



Figure 4.12: Incubator used for tests.

On the other hand, an important feature of the RFC is the programmable output power. It means that the range can be easily changed. If the communication paths are considered, this functionality proves to be important since it facilitates the transmission of the radio signals through obstacles. Table 4.1 shows the current consumption for some of the output power levels.

Table 4.1: Current consumption for output levels of the RFC [6].

Output Power (dBm)	Current Consumption (mA)
-55	8.4
-30	9.9
-2	17.7
0	21.2
+1	21.5

With these values it is possible to calculate the output power in Watts. Using the expression in 4.2.1, it is easy to convert the power in dBm to Volts, with R equal to 50 Ω which is the impedance of the antenna.

$$P_{dBm} = 10 \times \log \left(\frac{V^2}{R \times 0.001} \right) \quad (4.1)$$

For example, for +1 dBm the equation will return:

$$+1 = 10 \times \log \left(\frac{V^2}{50 \times 0.001} \right) \Leftrightarrow V = \sqrt[2]{10^{1/10} \times 50 \times 0.001} = 0.251V \quad (4.2)$$

Using this value and the current consumed taken from the previous table, it is possible to calculate the power in Watts.

$$P = V \times I \Leftrightarrow P = 0.251 \times 21.5 \times 10^{-3} \Leftrightarrow P = 5.4mW \quad (4.3)$$

For this test, the output power was programmed two times, one with +1 dBm, which is the maximum power, and with 0 dBm. Repeating the process for 0 dBm, the output power is 4.7 mW. This means that, at 0 dBm, the RFC will be using 87% of its maximum power.

Table 4.2 presents the conditions of the tests after merging the three scenarios of the path communication with the two scenarios of output power levels. It also contains the results of the tests when observing the range of the communications.

Table 4.2: Communication range for different scenarios.

Environment Test	Conditions	Output Power (mW)	Distance from BS to SB (meters)
Obstacle-free Path	None	4.7 (87%)	5.2
		5.4 (100%)	8.3
Incubator	Glass door (0.5 cm thickness)	4.7 (87%)	2.4
		5.4 (100%)	3.0
	Glass door + Metal door (7.5 cm total thickness)	4.7 (87%)	0.15
		5.4 (100%)	1.7
Walk-in freezer	Metal door (5 cm thickness)	4.7 (87%)	Not Working
		5.4 (100%)	0.9

4.2.2 Visual Interface

Once the VI is designed and integrated in the system, it needs to be tested for the five commands that need to be executed.

The first test consisted of testing for blinking of the LEDs which proved to be successful for all the scenarios related to device selection. The second test consisted of sending 4 bytes of data to the SB. The results for this test were positive for both devices and for broadcast mode. In order to test the correct writing of the data, the final test consisted of requesting the command 5 to each one of the SBs at a time.

After requesting the execution of commands 4 and 5 respectively, the data that was sent to write and received after reading is displayed in the VI as shown in figure 4.13

Select Device	Message Log	
<div>Device 1</div> <div>Device 2</div> <div>All Devices</div>	<div>Command "Read Data" was sent to Senso Board 1</div>	

Select Command	Data Payload	Description
Red LED		This Command will turn ON the Red LED on the Sensor Board(s) during 3 seconds
Orange LED		This Command will turn ON the Orange LED on the Sensor Board(s) during 3 seconds
Yellow LED		This Command will turn ON the Yellow LED on the Sensor Board(s) during 3 seconds
Write Data	Type Data: <input type="text" value="4"/> <input type="text" value="A"/> <input type="text" value="B"/> <input type="text" value="5"/>	This Command will send data to be written on the Sensor Board's flash memory
Read Data	Out data: <input type="text" value="4"/> <input type="text" value="A"/> <input type="text" value="B"/> <input type="text" value="5"/>	This Command will return the data read from the Sensor Board's flash memory

STOP

Figure 4.13: Demonstration of the execution of command 4 followed by command 5 in the Visual Interface.

Chapter 5

Conclusions

This thesis is based on the evolution of a wireless sensor network for monitoring pH and DO in chemical and biological processes. These processes are conducted under certain conditions, which in turn become system requirements for this thesis. The network consists of a single BS connected to the PC, and two SBs, responsible for monitoring pH and DO. Since this is a prototype, the network establishment and working is demonstrated using the blinking of three LEDs on the SB. Data can be sent to the SB flash memory and read from it as well.

The thesis revolves around the establishment of the WSN, addition of nodes (SBs) to the network, wireless communication within the network, and the development of a VI to aid the user in operation of the network.

5.1 Discussion

The implementation of this integrated system took place in a chemical laboratory, which added extra specific requirements to it. For example, it is required that the system works inside an incubator at a temperature of 37° C.

The wireless network protocol was implemented and tested for one BS and two SBs, both of them prototype boards. From the point of view of the BS, the communication can be established with the PC by using a USB port, or with the SBs via radio transmissions. As for the SB, it only communicates with the BS via radio. If one node drops out of the network, i.e. if for some reason it stops working, it is observed that the network functions with the remaining node normally. Thus, it can be said that the network is robust.

Five commands were implemented in the communication protocol, namely, blinking of the red, yellow, and orange LEDs assembled on the SBs, writing data into their flash memories, and reading data from the same. All these commands, except the reading data command, can be executed in the broadcast mode. In the broadcast mode, the BS transmits data to both the sensor nodes, i.e. both nodes are functional. The LEDs blink in a specific pattern, to visually illustrate the execution of these commands, for example, they blink once when data is sent, and twice when the computer requests to read the data.

These commands can be executed using the VI designed in *LabVIEW*. This will help the user request commands easily, without having to need knowledge regarding the working of the code. With its help, the user is able to select the SB(s) with which to communicate and select one of the commands implemented.

The network was tested under certain conditions to determine the distance ranges at which it worked. The boards were programmed at 87% and 100% of their output power in an obstacle-free path, sealed inside an incubator, and locked inside a freezer. The results lead to the conclusion that the higher the output power, the higher the transmission range. It may also conclude that, in the scenarios where there are obstacles between the SB and the BS, an increasing the output power will enable transmissions through metal.

5.2 Future Work

To be able to pass the prototype phase there are some future tasks that should be accomplished.

The improvement of the system's performance is achieved by adding hardware to the network. In the present case, the more SBs the system has, the more parameters can be measured in the bioprocess, and more bioprocesses can be monitored. Hence, it is very important to extend the network to a number of nodes, making it work with the maximum number of 16 nodes.

In this thesis, command 5, which is related to the reading process, was partially implemented, since the broadcast mode did not take place. This means that the data can not be read from both the boards simultaneously.

To make the VI more user-friendly more conditions should be added to the LabVIEW code. For example: displaying error messages if the board specified to write/read data is not in the network, including only those boards which are in the network in the list of devices.

To make the pH and DO measurements more suitable for a number of processes it is necessary to vary the brightness of the LEDs, i.e., vary the current flowing through them. To achieve this, the code written for the DAC needs to be reviewed. At this point, the control of this component is in binary mode. In other words, the pins are just being set and reset. In the future, the DAC should receive information about the amount of current to be supplied to the LEDs.

Looking at the big picture, this prototype needs to be a stand-alone device for actual pH and DO measurements.

Appendix A

Board Schematics

A.1 Base Station Schematic

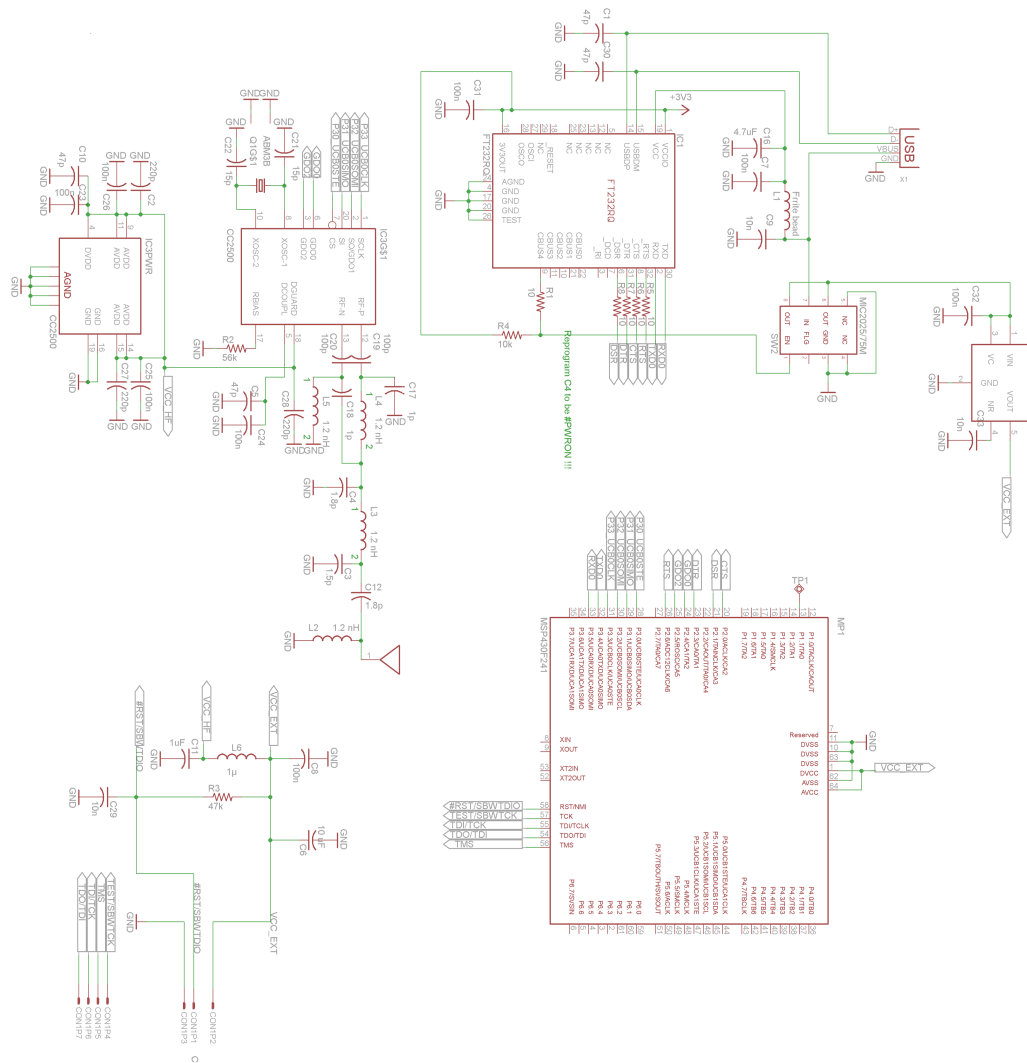


Figure A.2: Sensor board schematic.

Appendix B

Programming Codes

B.1 IAR Code

B.1.1 Base Station Code

B.1.1.1 transmit.c

```
////////////////////////////////////
//
// Description: This code is responsilbe for setting up the Base Station
//             as the Master of the Network.
//             Also enables the interface with the Computer via USB and
//             via Radio.
//
//
// Originally written by: Daniel Zurita  FEUP/UMBC  2014
////////////////////////////////////

#include "msp430f2471.h" // Microcontroller MSP430F2471 macros & defs
#include "stdint.h"      // MSP430 data type definitions
#include "wireless_functions.h" // Wireless setup and function definition

// GLOBAL VARIABLES INITIALIZATION
volatile unsigned char uart_rxd_command = 0; // command received from the computer
volatile unsigned char uart_rxd_destination = 0; // assigned position of the sensor board
                                                (1,2 or FF)
volatile unsigned char uart_data_to_send[4]; // data to be sent (used by command 4)
volatile unsigned char uart_count = 0;      // aux variable
volatile unsigned char cmd_status = 0x00;   // used to check if command was successfully sent
volatile unsigned char devices_connected = 0; // number of devices that joined the network
volatile unsigned char device_1 = 0;        // status of device 1 (0 - disconnected from
                                                network 1 - Connected to network)
volatile unsigned char device_2 = 0;        // status of device 2 (0 - disconnected from
                                                network 1 - Connected to network)
volatile unsigned char sensor_addr_1[2];    // used to save the address of the device 1
volatile unsigned char sensor_addr_2[2];    // used to save the address of the device 2

// FUNCTION THAT SENDS THE COMMAND TO BE EXECUTED BY THE SENSOR BOARD(S)
char Send_Command(char rxd_cmd , char rxd_dstn, volatile unsigned char *data_to_send)
{
```

```

static char command_packet[40] = {0x27,0x00,0x00,0x42,0x31,0x4E,0x36,0x43,0x00}; //
    formation of the command frame
if(rxd_dstn==0xFF)
{
    command_packet[1] = 0x46; // destination addr is set to "FF" (broadcast message)
    command_packet[2] = 0x46;
}
if(rxd_dstn==1 || rxd_dstn==2)
{
    command_packet[1] = 0x50; // destination addr is set to 'PX' with X being the device
        number
    command_packet[2] = 0x30 + rxd_dstn;
}

if(rxd_dstn!= 0x01 && rxd_dstn!= 0x02 && rxd_dstn!= 0xFF) // Check if destination is listed
{
    return 0x10; // Incorrect Destination
}

if(rxd_cmd!= 0x01 && rxd_cmd!= 0x02 && rxd_cmd!= 0x03 && rxd_cmd!= 0x04 && rxd_cmd!= 0x05)
    // Check if command is listed
{
    return 0x20; // Incorrect Command
}

command_packet[8] = 0x30+rxd_cmd;
command_packet[9] = data_to_send[0];
command_packet[10] = data_to_send[1];
command_packet[11] = data_to_send[2];
command_packet[12] = data_to_send[3];
command_packet[39] = 0x31; // Set END OF PACKET to 1 (0x31) If 1, packet is over, if 2
    packet is not over.

uint8_t pkt_lenght = sizeof(command_packet); // frame size

RFSendPacket(command_packet, pkt_lenght); // sends command to sensor board(s)

TI_CC_SPIStrobe(TI_CCxxx0_IDLE); // set cc2500 to Idle mode

if(rxd_cmd==0x05)
{
    return 0x50; // successfull command 5 transmission
}
else
{
    return 0x30; // successfull command 1,2,3 or 4 transmission
}
}

//FUNCTION FOR GLOBAL INITIALIZATION (FREQUENCY CLOCK / UART CONFIGURATION / CC2500 SETUP)
void Setup_Init(void)
{
    WDCTL = WDTW + WDTOLD; // stop watchdog

    volatile uint16_t delay;

    for(delay=0; delay<650; delay++); // let cc2500 Radio Chip settle on power up

```

```

// set up clock system
BCSCTL1 = CALBC1_8MHZ;          // set frequency clock
BCSCTL2 |= DIVS_3;              // SMCLK = MCLK/8 (1MHz)
DCOCTL = CALDCO_8MHZ;          // set MCLK to 8MHz

//UART CONFIGURATION (ENABLES COMMUNICATION WITH THE COMPUTER)
UCA0CTL1 = UCSWRST;
P3SEL = 0x30;                   // pins 3,4, 5 selected
UCA0CTL1 |= UCSSEL_2;           // SMCLK
UCA0BR0 = 8;                    // 1MHz 115200
UCA0BR1 = 0;                    // 1MHz 115200
UCA0MCTL |= (UCBRS2 + UCBRS1 + UCBRS0); // set UCBRF=0, UCBRS=7, UCOS16=0 (oversampling
                                     disabled)
IFG2 &= ~(UCA0RXIFG);
UCA0CTL1 &= ~UCSWRST;           // initializes USCI state machine
IE2 |= UCA0RXIE;                // enable USCI_A0 receiving interrupt

//WIRELESS INITIALIZATION

TI_CC_SPISetup();               // initialize SPI port for cc2500
P2SEL = 0;                      // sets P2.6 & P2.7 as GPIO
TI_CC_PowerupResetCCxxx();      // reset cc2500
writeRFSettings();              // write RF settings to config reg
TI_CC_GDO0_PxIES |= TI_CC_GDO0_PIN; // int on falling edge (end of pkt)
TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // clear flag
TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN; // enable int on end of packet
TI_CC_SPIStrobe(TI_CCxxx0_SRX); // initialize cc2500 in RX mode
TI_CC_SPIWriteReg(TI_CCxxx0_CHANNR, 0); // set Your Own Channel Number

for(delay=0; delay<650; delay++); // lets cc2500 Radio Chip finish setup
}

// FUNCTION THAT SENDS THE PERMISSION TO JOIN THE NETWORK
char Send_Network_Permission(void)
{
    static char grant_access[14] = {0x0D,0x00,0x00,0x42,0x31,0x4E,0x36,0x50,0x00,0x00,0x46,0x45
                                     ,0x55,0x50}; // formation of the grant-permission frame

    if(devices_connected==0)
    {
        grant_access[1] = sensor_addr_1[0]; // updates destination address using global variable
        grant_access[2] = sensor_addr_1[1];
        grant_access[8] = 0x31;              // position assigned to the device
    }
    if(devices_connected==1)
    {
        grant_access[1] = sensor_addr_2[0]; // updates destination address using global variable
        grant_access[2] = sensor_addr_2[1];
        grant_access[8] = 0x32;              // position assigned to the device
        grant_access[9] = 0x01;              // delay imposed by the network (seconds)
    }
    uint8_t pkt_lenght = sizeof(grant_access); // frame size
    RFSendPacket(grant_access, pkt_lenght); // Send request to join the network

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // set cc2500 to Idle mode

```



```

    return 0x80;
}

// INTERRUPT SERVICE ROUTINE (ISR) RESPONSIBLE FOR INTERFACING THE BASE STATION WITH THE
// COMPUTER (USING THE UART COMMUNICATION)
#pragma vector=USCIAB0RX_VECTOR
__interrupt void UART_RX_ISR(void)
{
    if(devices_connected>0) // checks if at least one SB is connected
    {
        //data collection from the LabVIEW
        if (UCA0RXIFG) // USCI_A0 RX buffer ready?
        {
            IFG2 &= ~UCA0RXIFG;
            if(uart_rxd_command==0)
            {
                uart_rxd_command = UCA0RXBUF; // saves command number
                return;
            }
            else if(uart_rxd_command!=0)
            {
                if(uart_rxd_destination==0)
                {
                    IFG2 &= ~UCA0RXIFG;
                    uart_rxd_destination = UCA0RXBUF; // saves destination address

                    if(uart_rxd_command==0x01 || uart_rxd_command==0x02 || uart_rxd_command==0x03 ||
                       uart_rxd_command==0x05) // checks if the command is 1, 2, 3 or 5
                    {
                        cmd_status=Send_Command(uart_rxd_command , uart_rxd_destination, uart_data_to_send);
                        if(cmd_status==0x30) // checks if the command was successfully sent
                        {
                            uart_rxd_command = 0; // variables reset
                            uart_rxd_destination = 0;
                            uart_data_to_send[0] = 0;
                            uart_data_to_send[1] = 0;
                            uart_data_to_send[2] = 0;
                            uart_data_to_send[3] = 0;
                            TI_CC_SPIStrobe(TI_CCxxx0_SRX); // sets cc2500 in RX mode
                        }
                        else if(cmd_status==0x50)
                        {
                            TI_CC_SPIStrobe(TI_CCxxx0_SRX); // sets cc2500 in RX mode
                            uart_rxd_command = 0; // variables reset
                            uart_rxd_destination = 0;
                        }
                    }
                }
                return;
            }
            else if(uart_rxd_destination!=0)
            {
                if(uart_rxd_command==0x04) // checks if the command is 4
                {
                    IFG2 &= ~UCA0RXIFG;
                    uart_data_to_send[uart_count] = UCA0RXBUF; // saves data so be sent to Sensor Board
                    uart_count += 1;
                }
            }
        }
    }
}

```

```

    if(uart_count==4) // is data over?
    {
        uart_count = 0; // variable reset
        cmd_status=Send_Command(uart_rxd_command , uart_rxd_destination, uart_data_to_send)
        ;
        if(cmd_status==0x30) // check if the command was successfully sent
        {
            uart_rxd_command = 0;          // variables reset
            uart_rxd_destination = 0;
            uart_data_to_send[0] = 0;
            uart_data_to_send[1] = 0;
            uart_data_to_send[2] = 0;
            uart_data_to_send[3] = 0;
            TI_CC_SPIStrobe(TI_CCxxx0_SRX); // sets cc2500 in RX mode
        }
    }
}

}

}

}

}

}

}

}

}

// INTERRUPT SERVICE ROUTINE (ISR) RESPONSIBLE FOR ACQUIRING DATA SENT FROM THE SENSOR BOARD(S
) (REQUEST TO JOIN THE NETWORK OR DATA SENT DUE TO EXECUTION OF THE COMMAND 5)
#pragma vector=PORT2_VECTOR
__interrupt void Port2_ISR(void)
{
    if(TI_CC_GDO0_PxIFG & TI_CC_GDO0_PIN)
    {
        uint8_t status[2];          // Buffer to store frame status bytes
        static uint8_t crcOk;      // flag variable to check if the received frame is good

        if(devices_connected>=0 && devices_connected<=2)
        {
            uint8_t len = 0x06;      // lenght of the frame that is intended to receive = 6 bytes
            uint8_t rx[6];           // buffer to store received frame
            uint8_t rx_2[39];
            crcOk = RFReceivePacket(rx,rx_2,&len,status); // fetch packet from cc2500

            if(crcOk==0x80) // condition to manage network
            {
                if(rx[2]==0x55 && rx[3]==0x4D && rx[4]==0x42 && rx[5]==0x43) // checks if password is "
                    UMBC "
                {
                    static uint8_t permission_granted = 0x00;

                    if(device_1==0) // if network is empty
                    {
                        sensor_addr_1[0]= rx[0]; // saves address from Sensor Board as first device
                        sensor_addr_1[1]= rx[1];
                        __delay_cycles(12000000); // 1.5 sec delay
                        permission_granted = Send_Network_Permission(); // sends permission to join the
                            network
                        device_1 = 1;
                    }
                }
            }
        }
    }
}

```

```

    if(device_2==0 && devices_connected ==1) // if first device already joined the network
    {
        sensor_addr_2[0]=rx[0]; // saves address from Sensor Board as second device
        sensor_addr_2[1]=rx[1];
        __delay_cycles(12000000); // 1.5 sec delay
        permission_granted = Send_Network_Permission();
        device_2 = 1;
    }
    if (permission_granted == 0x80)
    {
        devices_connected += 1; // update number of devices connected
    }
}
else if(crcOk==0x50) // condition to acquire data payload due to command 5 request
{
    int count = 0;

    while(count<4) // acquires data payload
    {
        while (!(IFG2&UCA0TXIFG));
        UCA0TXBUF = rx_2[6+count]; // sends data payload to the computer for further display
        count++;
    }

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // sets cc2500 to Idle mode
    TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // resets flag
    return;
}
}

TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // resets flag
if(devices_connected<2)
{
    TI_CC_SPIStrobe(TI_CCxxx0_SRX); // sets cc2500 to RX mode
}
}

// MAIN FUNCTION
void main()
{
    Setup_Init();

    __bis_SR_register(LPM4_bits + GIE); // sleep until next interrupt service routine (low power
        mode 4)
}

```

B.1.1.2 wireless_functions.h

```

////////////////////////////////////
// This file contains several functions that simplify interfacing the
// MSP430 software to the cc2500 transceiver chip. These funcs handle
// the details of cc2500 initialization and SPI communication.
//
// This file is edited down from the original file to be case-specific
// to the ez430-RF2500 target card.
//
// Original written: K. Quiring          TI, Inc 2006/07
// Brilliantly adapted: B. Ghena & A. Maine MTU/ECE 2012/04
// Revised and edited: R. Kieckhafer    MTU/ECE 2013/04
// Revised and re-adapted: P. Ramesh and R. Kieckhafer MTU/ECE 2014/04
//
// Final adaptation: Daniel Zurita      FEUP/UMBC 2014/07
////////////////////////////////////

#include "include_ez430.h" // Lots of #defines for cc2500 interfacing

char paTable[] = {0xFF}; // used in setting CC2500 power levels (set to maximum power)
char paTableLen = 1; // these were taken from helper files

////////////////////////////////////
// void TI_CC_Wait(unsigned int cycles)
//
// DESCRIPTION:
// Delay function, stalls for specified number of clock cycles.
// Actual num of clock cycles delayed is approx "cycles" argument. Specif,
// it's ((cycles-15) % 6) + 15. Not exact, but gives a sense of real-time
// delay. Also, if MCLK ~1MHz, "cycles" is approx num of usec delayed.
////////////////////////////////////
void TI_CC_Wait(unsigned int cycles)
{
    while(cycles>15)    // 15 cycles consumed by overhead
        cycles = cycles - 6; // 6 cycles consumed each iteration
}

////////////////////////////////////
// void TI_CC_SPISetup(void)
//
// DESCRIPTION: Configures the assigned interface to function as a SPI
// port and initializes it.
////////////////////////////////////
void TI_CC_SPISetup(void)
{
    TI_CC_CSn_PxOUT |= TI_CC_CSn_PIN;
    TI_CC_CSn_PxDIR |= TI_CC_CSn_PIN;    // Clear CSn
    UCB0CTL1 |= UCSWRST;    // Disab USCI state mach
    UCB0CTL0 |= UCMST+UCCKPH+UCMSB+UCSYNC; // 3-pin, 8-bit SPI mast
    UCB0CTL1 |= UCSSEL_2;    // SCLK = SMCLK
    UCB0BR0 = 0x02;    // UCLK/2
    UCB0BR1 = 0;
    TI_CC_SPI_USCIB0_PxSEL |= TI_CC_SPI_USCIB0_SIMO // SPI pins option sel
                          | TI_CC_SPI_USCIB0_SOMI
                          | TI_CC_SPI_USCIB0_UCLK;
}

```

```

TI_CC_SPI_USCIB0_PxDIR |= TI_CC_SPI_USCIB0_SIMO // SPI Tx pins out dir
                        | TI_CC_SPI_USCIB0_UCLK;

UCB0CTL1 &= ~UCSWRST;          // Enab USCI state mach
}

////////////////////////////////////
// void TI_CC_SPIWriteReg(char addr, char value)
//
// DESCRIPTION:
// Writes "value" to a single config. register at address "addr".
////////////////////////////////////
void TI_CC_SPIWriteReg(char addr, char value)
{
    TI_CC_CSxOUT &= ~TI_CC_CSx_PIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG));     // Wait for TXBUF ready
    UCB0TXBUF = addr;                 // Send reg address
    while (!(IFG2 & UCB0TXIFG));     // Wait for TXBUF ready
    UCB0TXBUF = value;                // Send data
    while (UCB0STAT & UCBUSY);        // Wait for SPI done
    TI_CC_CSxOUT |= TI_CC_CSx_PIN;    // Clear CSx
}

////////////////////////////////////
// void TI_CC_SPIWriteBurstReg(char addr, char *buffer, char count)
//
// DESCRIPTION:
// Writes values to multiple config. registers, the first register being
// at address "addr". First data byte is at "buffer", and both addr and
// buffer are incremented sequentially (within the CCxxx & MSP430,
// respectively) until "count" writes have been performed.
////////////////////////////////////
void TI_CC_SPIWriteBurstReg(char addr, char *buffer, char count)
{
    unsigned int i;
    TI_CC_CSxOUT &= ~TI_CC_CSx_PIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG));     // Wait for TXBUF ready
    UCB0TXBUF = addr | TI_CC_CSx0_WRITE_BURST; // Send reg address
    for (i = 0; i < count; i++)
    {
        while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
        UCB0TXBUF = buffer[i];       // Send data
    }
    while (UCB0STAT & UCBUSY);        // Wait for SPI done
    TI_CC_CSxOUT |= TI_CC_CSx_PIN;    // Clear CSx
}

////////////////////////////////////
// char TI_CC_SPIReadReg(char addr)
//
// DESCRIPTION:
// Reads a single config. register at address "addr" & returns the
// value read.
////////////////////////////////////
char TI_CC_SPIReadReg(char addr)
{
    char x;

```

```

TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
UCB0TXBUF = (addr | TI_CCxxx0_READ_SINGLE); // Send reg address
while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
UCB0TXBUF = 0; // Dummy Wr so we can read
while (UCB0STAT & UCBUSY); // Wait for SPI done
x = UCB0RXBUF; // Read data
TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
return x;
}

// void TI_CC_SPIReadBurstReg(char addr, char *buffer, char count)
//
// DESCRIPTION:
// Reads multiple config. registers, the first register being at address
// "addr". Values read are deposited sequentially starting at address
// "buffer", until "count" registers have been read.
//
void TI_CC_SPIReadBurstReg(char addr, char *buffer, char count)
{
    char i;
    TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
    UCB0TXBUF = (addr | TI_CCxxx0_READ_BURST); // Send reg address
    while (UCB0STAT & UCBUSY); // Wait for SPI done
    UCB0TXBUF = 0; // Dummy Wr to read 1st byte

    // Addr byte is now being TX'ed, with dummy byte to follow immedi. after

    IFG2 &= ~UCB0RXIFG; // Clear IRQ flag
    while (!(IFG2 & UCB0RXIFG)); // Wait for end of 1st data byte
    // First data byte now in RXBUF

    for (i = 0; i < (count-1); i++)
    {
        UCB0TXBUF = 0; //Initiate next data RX, meanwhile.
        buffer[i] = UCB0RXBUF; // Store data from last data RX
        while (!(IFG2 & UCB0RXIFG)); // Wait for SPI done
    }
    buffer[count-1] = UCB0RXBUF; // Store last RX byte in buffer
    TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
}

// char TI_CC_SPIReadStatus(char addr)
//
// DESCRIPTION:
// Special function for reading status registers. Reads status register
// at register "addr" & returns the value read.
//
char TI_CC_SPIReadStatus(char addr)
{
    char status;
    TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
    UCB0TXBUF = (addr | TI_CCxxx0_READ_BURST); // Send reg address
    while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready

```

```

UCB0TXBUF = 0; // Dummy Wr so can read data
while (UCB0STAT & UCBUSY); // Wait for SPI done
status = UCB0RXBUF; // Read data
TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
return status;
}

// void TI_CC_SPIStrobe(char strobe)
//
// DESCRIPTION:
// Special function for writing to command strobe registers. Writes
// to the strobe at address "addr".
//
void TI_CC_SPIStrobe(char strobe)
{
    TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
    UCB0TXBUF = strobe; // Send strobe
    // Strobe addr is now being TX'ed
    while (UCB0STAT & UCBUSY); // Wait for SPI done
    TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
}

// void TI_CC_PowerupResetCCxxxx(void)
//
// DESCRIPTION:
// Function for resetting the CCxxxx on powerup.
//
void TI_CC_PowerupResetCCxxxx(void)
{
    TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
    TI_CC_Wait(30); // Stall
    TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
    TI_CC_Wait(30); // Stall
    TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
    TI_CC_Wait(45); // Stall
    TI_CC_CSxOUT &= ~TI_CC_CSxPIN; // Assert CSx
    while (!(IFG2 & UCB0TXIFG)); // Wait for TXBUF ready
    UCB0TXBUF = TI_CCxxx0_SRES; // Send reset strobe
    // Strobe addr is now being TX'ed
    while (UCB0STAT & UCBUSY); // Wait for SPI done
    TI_CC_CSxOUT |= TI_CC_CSxPIN; // Clear CSx
}

// void writerFSettings(void)
//
// DESCRIPTION:
// Used to configure the CCxxxx registers.
//
// ARGUMENTS:
// none
//
// Product = CC2500
// Crystal accuracy = 40 ppm

```

```

// X-tal frequency = 26 MHz
// RF output power = 0 dBm
// RX filterbandwidth = 542 kHz RMK-2014-04 - iaw SmartRF Studio
// Deviation = 0.000000
// Return state: Return to RX state upon leaving either TX or RX
// Datarate = 250 kbps
// Modulation = (7) MSK
// Manchester enable = (0) Manchester disabled
// RF Frequency = 2423.5 MHz RMK-2014-04 Start of Wifi chnl 1-6 Gap
// Channel spacing = 250 kHz RMK-2014-04 Enlarged to lower interf.
// Channel number = ??
// Optimization = Sensitivity
// Sync mode = (3) 30/32 sync word bits detected
// Format of RX/TX data = (0) Normal mode, use FIFOs for RX & TX
// CRC operation = (1) CRC calculation in TX & CRC check in RX enabled
// Forward Error Correction = (0) FEC disabled
// Length config. = (1):
// => Variable length packets,
// => packet length configured by first received byte after sync word.
// Packetlength = 255
// Preamble count = (2) 4 bytes
// Append status = 1
// Address check = (0) No address check
// FIFO autoflush = 0
// Device address = 0
// GDO0 signal selection = (6):
// => Asserts when sync word has been sent / received,
// => & de-asserts at the end of the packet
// GDO2 signal selection = (11) Serial Clock
////////////////////////////////////
void writeRFSettings(void)
{
    // Write register settings
    TI_CC_SPIWriteReg(TI_CCxxx0_IOCFG2, 0x0B); // GDO2 out pin config
    TI_CC_SPIWriteReg(TI_CCxxx0_IOCFG0, 0x06); // GDO0 out pin config
    TI_CC_SPIWriteReg(TI_CCxxx0_PKTLEN, 0xFF); // Packet length
    TI_CC_SPIWriteReg(TI_CCxxx0_PKTCTRL1, 0x04); // Packet automation ctrl
    TI_CC_SPIWriteReg(TI_CCxxx0_PKTCTRL0, 0x05); // Packet automation ctrl
    TI_CC_SPIWriteReg(TI_CCxxx0_ADDR, 0xFF); // Device address.
    TI_CC_SPIWriteReg(TI_CCxxx0_CHANNR, 0x59); // Channel number
    TI_CC_SPIWriteReg(TI_CCxxx0_FSCTRL1, 0x0A); // Freq synth. ctrl - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_FSCTRL0, 0x00); // Freq synth. ctrl
    TI_CC_SPIWriteReg(TI_CCxxx0_FREQ2, 0x5D); // Freq ctrl word hi byte
    TI_CC_SPIWriteReg(TI_CCxxx0_FREQ1, 0x36); // Freq ctrl word mid - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_FREQ0, 0x27); // Freq ctrl word low - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_MDMCFG4, 0x2D); // Modem config.
    TI_CC_SPIWriteReg(TI_CCxxx0_MDMCFG3, 0x3B); // Modem config.
    TI_CC_SPIWriteReg(TI_CCxxx0_MDMCFG2, 0x73); // Modem config. - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_MDMCFG1, 0x23); // Modem config. - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_MDMCFG0, 0x3B); // Modem config. - RMK
    TI_CC_SPIWriteReg(TI_CCxxx0_DEVIATN, 0x00); // Modem dev (FSK only)
    TI_CC_SPIWriteReg(TI_CCxxx0_MCSM1, 0x3F); // MainRad Ctrl stat mach
    TI_CC_SPIWriteReg(TI_CCxxx0_MCSM0, 0x18); // MainRad Ctrl stat mach
    TI_CC_SPIWriteReg(TI_CCxxx0_FOCCFG, 0x1D); // Freq Offset Comp Config
    TI_CC_SPIWriteReg(TI_CCxxx0_BSCFG, 0x1C); // Bit synch. config.
    TI_CC_SPIWriteReg(TI_CCxxx0_AGCCTRL2, 0xC7); // AGC ctrl.
    TI_CC_SPIWriteReg(TI_CCxxx0_AGCCTRL1, 0x00); // AGC ctrl.

```



```

////////////////////////////////////
// char RfReceivePacket(char *rxBuffer, char *rxBuffer_2, char *length, char *status)
//
// DESCRIPTION:
// Receives a packet of variable length (first byte in packet must be the
// length byte). The packet length should not exceed the RXFIFO size.
//
// RXBYTES register is first read to ensure there are bytes in the FIFO.
// This is done because GDO signal will go high even if FIFO is flushed
// due to address filtering, CRC filtering, or packet length filtering.
//
// ARGUMENTS:
// char *rxBuffer = Ptr to buffer where incoming data should be stored (when receiving
// request to join the network)
// char *rxBuffer_2 = Ptr to buffer where incoming data should be stored (when receiving
// answer after execution of command 5)
// char *length = Ptr to a variable containing the size of the buffer
// where incoming data should be stored. After this func
// returns, that variable holds the packet length.
// char *status = Ptr to store the returned status of the receive.
// Contains RSSI & LQI.
//
// RETURN VALUE:
// char = 0x80: CRC OK indicates that a request was received
// = 0x50: CRC OK indicates that data from the SB's flash memory was received
// = 0x00: CRC NOT OK (or no pkt was put in RXFIFO due to filtering)
////////////////////////////////////
char RfReceivePacket(char *rxBuffer, char *rxBuffer_2, char *length, char *status)
{
    char pktLen;
    if ((TI_CC_SPIReadStatus(TI_CCxxx0_RXBYTES) & TI_CCxxx0_NUM_RXBYTES))
    {
        pktLen = TI_CC_SPIReadReg(TI_CCxxx0_RXFIFO); // Read length byte
        if (pktLen <= *length) // If pktLen <= rxBuffer
        {
            // Then save packet
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, rxBuffer, pktLen); // Get data
            *length = pktLen; // Get Len
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, status, 2); // Get stat
            return (char) (status[TI_CCxxx0_LQI_RX]&TI_CCxxx0_CRC_OK); // Ret OK
        }
        else if (pktLen==0x27)
        {
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, rxBuffer_2, pktLen); // Get data
            *length = pktLen; // Get Len
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, status, 2); // Get stat
            return 0x50; // Ret OK
        }
        else
        {
            *length = pktLen; // Return the large size
            TI_CC_SPIStrobe(TI_CCxxx0_SFRX); // Flush RXFIFO
            return 0; // Retn Error flag
        }
    }
    else
    {
        return 0; // Retn Error flag
    }
}

```

B.1.2 Sensor Board Code

B.1.2.1 receive.c

```

////////////////////////////////////
//
// Description: This code is responsilbe for setting up the Sensor Board
//             as a Slave of the Network.
//             Also enables the interface with the Base Station via Radio
//
//
// Originally written by: Daniel Zurita  FEUP/UMBC  2014
//
////////////////////////////////////

#include "msp430f2272.h"    // Microcontroller MSP430F2272 macros & defs
#include "stdint.h"         // MSP430 data type definitions
#include "wireless_functions.h" // wireless setup and function definition

// GLOBAL VARIABLES INITIALIZATION
volatile unsigned char nw_addr[2] = {0x4E,0x00}; // network address. set to default before
                                                joining any network
volatile unsigned char device_status = 0; // status of the device (0 - disconnected from
                                           network 1 - Connected to network)
volatile unsigned char base_station_addr[2]; // used to save the base station address
volatile unsigned char position_number[2]; // used to save the assigned position
volatile unsigned char delay_to_answer[1]; // used to save the delay of operation
volatile unsigned char command[2];         // used to save the command that will be executed
volatile unsigned char data_payload_rxd[4]; // used to save the received data on the flash
                                           memory
volatile unsigned char data_payload_txd[4]; // used to send the data read from the flash
                                           memory

// LED SETUP FUNCTIONS
void Reset_LEDs(void)
{
    P4OUT |= 0x40; // 4.6 set to high!
    P4OUT &= ~0x20; // 4.5 set to low!
    P4OUT &= ~0x07; // 4.0,1,2 set to low!
    P4DIR |= 0x67; // pins 4.0, 4.1, 4.2, 4.5 and 6 as output
}

void Set_RedLED(void)
{
    P4OUT |= 0x01; // 4.0 set to high! LED
    P4OUT &= ~0x40; // 4.6 set to low!
    P4OUT |= 0x20; // 4.5 set to high!
}

void Set_YellowLED(void)
{
    P4OUT |= 0x02; // 4.1 set to high! LED
    P4OUT &= ~0x40; // 4.6 set to low!
    P4OUT |= 0x20; // 4.5 set to high!
}

```

```

void Set_OrangeLED(void)
{
    P4OUT |= 0x04; // 4.2 set to high! LED
    P4OUT &= ~0x40; // 4.6 set to low!
    P4OUT |= 0x20; // 4.5 set to high!
}

//GLOBAL INITIALIZATION FUNCTION
void Setup_Init(void)
{
    WDTCTL = WDTPW + WDTHOLD;    // stop watchdog

    volatile uint16_t delay;

    for(delay=0; delay<650; delay++); // let cc2500 Radio Chip settle on power up

    // set up clock system
    BCSCTL1 = CALBC1_8MHZ;        // set frequency clock
    BCSCTL2 |= DIVS_3;            // SMCLK = MCLK/8 (1MHz)
    DCOCTL = CALDCO_8MHZ;        // set MCLK to 8MHz

    // enable config for LED
    P4SEL = 0;                    // all pins of port 4 are selected
    P3SEL = 0x30;                 // pins 3.4, 5 selected
    UCA0CTL1 |= UCSSEL_2;
    UCA0BR0 = 104;                // 1MHz 9600 baud rate
    UCA0BR1 = 0;                  // 1MHz 9600 baud rate
    UCA0MCTL = UCBRS0;            // modulation UCBRSx = 1
    UCA0CTL1 &= ~UCSWRST;        // initializes USCI state machine

    // wireless Initialization
    TI_CC_SPISetup();             // initialize SPI port for cc2500
    P2SEL = 0;                    // sets P2.6 & P2.7 as GPIO
    TI_CC_PowerupResetCCxxxx();  // reset cc2500
    writeRFSettings();            // write RF settings to config reg
    TI_CC_GDO0_PxIES |= TI_CC_GDO0_PIN; // int on falling edge (end of pkt)
    TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // clear flag
    TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN; // enable int on end of packet
    TI_CC_SPIStrobe(TI_CCxxx0_SRX); // initialize cc2500 in RX mode.
    TI_CC_SPIWriteReg(TI_CCxxx0_CHANNR, 0); // set Your Own Channel Number

    for(delay=0; delay<650; delay++); // lets cc2500 Radio Chip finish setup
}

// FLAHS MEMORY CLOCK SETUP FUNCTION
void Setup_Flash_Clock(void)
{
    DCOCTL = 0;                    // select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_1MHZ;        // set DCO to 1MHz
    DCOCTL = CALDCO_1MHZ;
    FCTL2 = FWKEY + FSSEL0 + FN1; // MCLK/3 for Flash Timing Generator
}

// FLAHS MEMORY WRITE FUNCTION
char Write_Flash_SegC(void)
{

```

```

char *Flash_ptr;           // flash pointer
unsigned int i = 0;

Flash_ptr = (char *)0x1040; // initialize flash pointer to segment C
FCTL3 = FWKEY;             // clear Lock bit
FCTL1 = FWKEY + ERASE;     // set Erase bit
*Flash_ptr = 0;            // dummy write to erase flash seg
FCTL1 = FWKEY + WRT;       // set WRT bit for write operation

while(i<4)
{
    *Flash_ptr++ = data_payload_rxd[i]; // write value to flash
    i++;
}

FCTL1 = FWKEY;             // clear WRT bit
FCTL3 = FWKEY + LOCK;     // set LOCK bit

return 0x10;
}

// FLAHS MEMORY READ FUNCTION
char Read_Flash_SegC(void)
{
    char *Flash_ptr;       // flash pointer
    unsigned int i = 0;

    Flash_ptr = (char *)0x1040; // initialize flash pointer to segment C

    while(i<4)
    {
        data_payload_txd[i] = *Flash_ptr++; // write value to flash
        i++;
    }

    return 0x10;
}

// FUNCTION THAT SENDS THE REQUEST TO THE BASE STATION TO JOIN THE NETWORK
void Join_Network(void)
{
    static char join_nw[7] = {0x06,0x53,0x32,0x55,0x4D,0x42,0x43}; // formation of the request
    frame
    uint8_t pkt_lenght = sizeof(join_nw); // frame size
    RFSendPacket(join_nw, pkt_lenght);    // sends request to join the network
    TI_CC_SPIStrobe(TI_CCxxx0_IDLE);      // set cc2500 to idle mode
    TI_CC_SPIStrobe(TI_CCxxx0_SRX);       // set CC2500 to receive mode (waiting for
    answer)
}

// FUNCTION THAT SENDS THE DATA READ FROM THE FLASH MEMORY TO THE BASE STATION
char Send_Data(void)
{
    static char data_packet[40] = {0x27,0x00,0x00,0x00,0x00,0x4E,0x00}; // formation of the
    frame to send data

```

```

data_packet[1] = base_station_addr[0];           // updates destination address using
    global variable
data_packet[2] = base_station_addr[1];
data_packet[3] = position_number[0];             // updates source address using
    global variable
data_packet[4] = position_number[1];
data_packet[6] = nw_addr[1];                     // updates network address using
    global variable
data_packet[7] = data_payload_txd[0];            // updates data read from flash using
    global variable
data_packet[8] = data_payload_txd[1];
data_packet[9] = data_payload_txd[2];
data_packet[10] = data_payload_txd[3];
data_packet[39] = 0x31;                          // updates end of frame byte

uint8_t pkt_lenght = sizeof(data_packet);        // frame size
RFSendPacket(data_packet, pkt_lenght);           // sends data to base station

return 0x50;
}

// INTERRUPT SERVICE ROUTINE (ISR) RESPONSIBLE FOR ACQUIRING DATA SENT FROM THE BASE STATION
#pragma vector=PORT2_VECTOR
__interrupt void PktRxedISR(void)
{
    if(TI_CC_GDO0_PxIFG & TI_CC_GDO0_PIN)
    {
        uint8_t status[2];                      // buffer to store frame status bytes
        static uint8_t crcOk;                   // flag variable to check if the received frame is good

        if(device_status==0)                   // check if device is not connected to the Network
        {
            uint8_t len = 0x0D;                 // lenght of the frame that is intended to receive = 13
            bytes
            uint8_t rx[13];                     // buffer to store received frame

            crcOk = RFReceivePacket(rx,&len,status); // fetch packet from cc2500

            if(crcOk)
            {
                if(rx[9]==0x46 && rx[10]==0x45 && rx[11]==0x55 && rx[12]==0x50) // checks if password
                    is " FEUP "
                {
                    if(rx[0]==0x53 && rx[1]==0x32) // check if the message was sent for
                        this sensor board (this case it is Sensor Board 2)
                    {
                        base_station_addr[0]= rx[2]; // saves baste station address
                        base_station_addr[1]= rx[3];
                        nw_addr[0] = rx[4];          // saves newtwork address
                        nw_addr[1] = rx[5];
                        position_number[0] = rx[6]; // saves newtwork address
                        position_number[1] = rx[7];
                        delay_to_answer[0] = rx[8]; // saves delay information
                        device_status = 1;           // updates status of the device to 1 (
                            connected)

                        int ok = 1;

```

```

    while(ok<=3)                                // blinks the red LED 3 times
        indicating that the communication was established
    {
        Reset_LEDs();
        Set_RedLED();
        __delay_cycles(2000000);                // 1/4 second delay
        Reset_LEDs();
        __delay_cycles(2000000);                // 1/4 second delay
        ok++;
    }
}
}
return;
}
}

if(device_status==1)                            // check if device is connected to the Network
{
    uint8_t len = 0x27;                          // lenght of the frame that is intended to receive =
        39 bytes
    uint8_t rx_cmd[39];                          // buffer to store received frame

    crcOk = RFReceivePacket(rx_cmd,&len,status); // fetch packet from cc2500

    if(crcOk)
    {
        if((rx_cmd[0]==position_number[0] && rx_cmd[1]==position_number[1]) || (rx_cmd[0]==0x46
            && rx_cmd[1]==0x46)) // checks if the message was sent for this board or a broadcast
        {
            if(rx_cmd[2]==base_station_addr[0] && rx_cmd[3]==base_station_addr[1]) // check if the
                message was sent from the base station responsible for the network
            {
                if(rx_cmd[4]==nw_addr[0] && rx_cmd[5]==nw_addr[1]) // check if the network corresponds
                    to the one which this board is connected to
                {
                    command[0] = rx_cmd[6]; // saves command received from base station
                    command[1] = rx_cmd[7];

                    if(command[1]==0x34) // if the command to be executed is the command 4
                    {
                        int count = 0;
                        static uint8_t write_status = 0; // status of the writting process.

                        while(count<4)
                        {
                            data_payload_rxd[count] = rx_cmd[8+count]; // saves data payload received from
                                base estation
                            count++;
                        }

                        Setup_Flash_Clock();
                        write_status = Write_Flash_SegC(); // writes data on the flash memory

                        if(write_status) // if the writting was succesful, the red and the orange LEDs will
                            blink once for 1/2 second
                        {

```

```

    Setup_Init();
    Set_RedLED();
    Set_OrangeLED();
    __delay_cycles(4000000); // 1/2 second delay
    Reset_LEDs();
}
}

if(command[1]==0x35) // if the command to be executed is the command 5
{
    static uint8_t read_status = 0;
    static uint8_t read_data = 0;
    read_status = Read_Flash_SegC(); // reads data from flash memory

    if(read_status)
    {
        read_data = Send_Data(); // sends data from flash memory to the base station

        if(read_data) // if data was properly sent the red and the orange
            LEDs will blink twice
        {
            Set_RedLED();
            Set_OrangeLED();
            __delay_cycles(1000000); // 1/8 second delay
            Reset_LEDs();
            __delay_cycles(1000000); // 1/8 second delay
            Set_RedLED();
            Set_OrangeLED();
            __delay_cycles(1000000); // 1/8 second delay
            Reset_LEDs();
        }
    }
}

if(rx_cmd[0]==0x46 && rx_cmd[1]==0x46 && position_number[1]==0x32)
{
    __delay_cycles(8000000); // 1 second delay
    Reset_LEDs();
}

if(command[1]==0x31) // if the command to be executed is the command 1
{
    Set_RedLED();
    __delay_cycles(24000000); // 3 second delay
    Reset_LEDs();
}

if(command[1]==0x32) // if the command to be executed is the command 2
{
    Set_YellowLED();
    __delay_cycles(24000000); // 3 second delay
    Reset_LEDs();
}

if(command[1]==0x33) // if the command to be executed is the command 3
{
    Set_OrangeLED();

```



```

        __delay_cycles(24000000); // 3 second delay
        Reset_LEDs();
    }
}
}
}
}

TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // seset IRQ flag
TI_CC_SPIStrobe(TI_CCxxx0_SIDLE); // set cc2500 to idle mode.
TI_CC_SPIStrobe(TI_CCxxx0_SRX); // set cc2500 to RX mode.
}

// MAIN FUNCTION
void main()
{

    Setup_Init();
    Reset_LEDs();

    Join_Network();
    __bis_SR_register(LPM4_bits + GIE); // sleep until next interrupt service routine (low power
        mode 4)
}

```

B.1.2.2 wireless_functions.h

This file is similar to the one in section [B.1.1.2](#), except for the function given.

```

////////////////////////////////////
// char RFReceivePacket(char *rxBuffer, char *length, char *status)
//
// DESCRIPTION:
// Receives a packet of variable length (first byte in packet must be the
// length byte). The packet length should not exceed the RXFIFO size.

// To use this func, APPEND_STATUS in PKTCTRL1 register must be enabled.
// Assumes that the func is called after it is known that a packet has
// been received; for example, in response to GDO0 going low when it is
// configured to output packet reception status.
//
// RXBYTES register is first read to ensure there are bytes in the FIFO.
// This is done because GDO signal will go high even if FIFO is flushed
// due to address filtering, CRC filtering, or packet length filtering.
//
// ARGUMENTS:
// char *rxBuffer = Ptr to buffer where incoming data should be stored
// char *length = Ptr to a variable containing the size of the buffer
//                where incoming data should be stored. After this func
//                returns, that variable holds the packet length.
// char *status = Ptr to store the returned status of the receive.
//                Contains RSSI & LQI.
//
// RETURN VALUE:
// char = 0x80: CRC OK
//        = 0x00: CRC NOT OK (or no pkt was put in RXFIFO due to filtering)
////////////////////////////////////
char RFReceivePacket(char *rxBuffer, char *length, char *status)
{
    char pktLen;
    if ((TI_CC_SPIReadStatus(TI_CCxxx0_RXBYTES) & TI_CCxxx0_NUM_RXBYTES))
    {
        pktLen = TI_CC_SPIReadReg(TI_CCxxx0_RXFIFO); // Read length byte
        if (pktLen <= *length) // If pktLen <= rxBuffer
        {
            // Then save packet
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, rxBuffer, pktLen); // Get data
            *length = pktLen; // Get Len
            TI_CC_SPIReadBurstReg(TI_CCxxx0_RXFIFO, status, 2); // Get stat
            return (char)(status[TI_CCxxx0_LQI_RX]&TI_CCxxx0_CRC_OK); // Ret OK
        }
        else
        {
            *length = pktLen; // Return the large size
            TI_CC_SPIStrobe(TI_CCxxx0_SFRX); // Flush RXFIFO
            return 0; // Retn Error flag
        }
    }
    else
    {
        return 0; // Return Error flag
    }
}

```

B.2 Include Files

B.2.1 stdint.h

This file includes data type definitions for the μ Cs.

```
#define __USING_MINT8 0
#define __CONCATenate(left, right) left ## right
#define __CONCAT(left, right) __CONCATenate(left, right)

#define INT8_MAX 0x7f
#define INT8_MIN (-INT8_MAX - 1)
#define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
#define INT16_MAX 0x7fff
#define INT16_MIN (-INT16_MAX - 1)
#define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
#define INT32_MAX 0x7fffffffL
#define INT32_MIN (-INT32_MAX - 1L)
#define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
#define INT64_MAX 0x7fffffffffffffffLL
#define INT64_MIN (-INT64_MAX - 1LL)
#define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)

#define INT_LEAST8_MAX INT8_MAX
#define INT_LEAST8_MIN INT8_MIN
#define UINT_LEAST8_MAX UINT8_MAX
#define INT_LEAST16_MAX INT16_MAX
#define INT_LEAST16_MIN INT16_MIN
#define UINT_LEAST16_MAX UINT16_MAX
#define INT_LEAST32_MAX INT32_MAX
#define INT_LEAST32_MIN INT32_MIN
#define UINT_LEAST32_MAX UINT32_MAX
#define INT_LEAST64_MAX INT64_MAX
#define INT_LEAST64_MIN INT64_MIN
#define UINT_LEAST64_MAX UINT64_MAX

#define INT_FAST8_MAX INT8_MAX
#define INT_FAST8_MIN INT8_MIN
#define UINT_FAST8_MAX UINT8_MAX
#define INT_FAST16_MAX INT16_MAX
#define INT_FAST16_MIN INT16_MIN
#define UINT_FAST16_MAX UINT16_MAX
#define INT_FAST32_MAX INT32_MAX
#define INT_FAST32_MIN INT32_MIN
#define UINT_FAST32_MAX UINT32_MAX
#define INT_FAST64_MAX INT64_MAX
#define INT_FAST64_MIN INT64_MIN
#define UINT_FAST64_MAX UINT64_MAX

#define INTPTR_MAX INT16_MAX
#define INTPTR_MIN INT16_MIN
#define UINTPTR_MAX UINT16_MAX

#define INTMAX_MAX INT64_MAX
#define INTMAX_MIN INT64_MIN
#define UINTMAX_MAX UINT64_MAX
```

```

#define PTRDIFF_MAX INT16_MAX
#define PTRDIFF_MIN INT16_MIN
#define SIG_ATOMIC_MAX INT8_MAX
#define SIG_ATOMIC_MIN INT8_MIN
#define SIZE_MAX (__CONCAT(INT16_MAX, U))

#define INT8_C(value) ((int8_t) value)
#define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
#define INT16_C(value) value
#define UINT16_C(value) __CONCAT(value, U)
#define INT32_C(value) __CONCAT(value, L)
#define UINT32_C(value) __CONCAT(value, UL)
#define INT64_C(value) __CONCAT(value, LL)
#define UINT64_C(value) __CONCAT(value, ULL)
#define INTMAX_C(value) __CONCAT(value, LL)
#define UINTMAX_C(value) __CONCAT(value, ULL)

typedef signed char int8_t;
typedef unsigned char uint8_t;
typedef signed int int16_t;
typedef unsigned int uint16_t;
typedef signed long int int32_t;
typedef unsigned long int uint32_t;
typedef signed long long int int64_t;
typedef unsigned long long int uint64_t;

typedef int16_t intptr_t;
typedef uint16_t uintptr_t;

typedef int8_t int_least8_t;
typedef uint8_t uint_least8_t;
typedef int16_t int_least16_t;
typedef uint16_t uint_least16_t;
typedef int32_t int_least32_t;
typedef uint32_t uint_least32_t;
typedef int64_t int_least64_t;
typedef uint64_t uint_least64_t;

typedef int8_t int_fast8_t;
typedef uint8_t uint_fast8_t;
typedef int16_t int_fast16_t;
typedef uint16_t uint_fast16_t;
typedef int32_t int_fast32_t;
typedef uint32_t uint_fast32_t;
typedef int64_t int_fast64_t;
typedef uint64_t uint_fast64_t;

typedef int64_t intmax_t;
typedef uint64_t uintmax_t;

```

B.2.2 include_ez430.h

This file was adapted to be implemented with the μ Cs used in the boards.

```

////////////////////////////////////
// Contains several #defines for cc2500 transceiver chip interfacing.
//
// This file is edited down from the original file to be case-specific
// to the ez430-RF2500 target card.
//
// Original written: K. Quiring, TI, Inc, 2006/07
// Brilliantly adapted: B. Ghena & A. Maine, MTU/ECE, 2012/04
// Revised & edited: R. Kieckhafer MTU/ECE, 2013/04
// Revised: Daniel Zurita FEUP/UMBC 2013/04
////////////////////////////////////

#include "msp430f2272.h" // microcontroller MSP430f2272 #defines file

// USCIB0 for ez430-RF2500 card
//-----
#define TI_CC_SPI_USCIB0_PxSEL P3SEL
#define TI_CC_SPI_USCIB0_PxDIR P3DIR
#define TI_CC_SPI_USCIB0_PxIN P3IN
#define TI_CC_SPI_USCIB0_SIMO BIT1
#define TI_CC_SPI_USCIB0_SOMI BIT2
#define TI_CC_SPI_USCIB0_UCLK BIT3

// Consts used to identify the chosen SPI & UART interfaces.
//-----
#define TI_CC_SER_INTF_NULL 0
#define TI_CC_SER_INTF_USART0 1
#define TI_CC_SER_INTF_USART1 2
#define TI_CC_SER_INTF_USCIA0 3
#define TI_CC_SER_INTF_USCIA1 4
#define TI_CC_SER_INTF_USCIA2 5
#define TI_CC_SER_INTF_USCIA3 6
#define TI_CC_SER_INTF_USCIB0 7 // MSP430 on ez430-RF2500 card
#define TI_CC_SER_INTF_USCIB1 8
#define TI_CC_SER_INTF_USCIB2 9
#define TI_CC_SER_INTF_USCIB3 10
#define TI_CC_SER_INTF_USI 11
#define TI_CC_SER_INTF_BITBANG 12

// MSP430 pin definitions and configuration aliases
//-----

#define TI_CC_LED_PxOUT P1OUT // LED pins
#define TI_CC_LED_PxDIR P1DIR
#define TI_CC_LED1 BIT0
#define TI_CC_LED2 BIT1

#define TI_CC_SW_PxIN P1IN // pushbutton pin
#define TI_CC_SW_PxIE P1IE
#define TI_CC_SW_PxIES P1IES
#define TI_CC_SW_PxIFG P1IFG
#define TI_CC_SW_PxREN P1REN

```

```

#define TI_CC_SW_PxOUT P1OUT
#define TI_CC_SW1 BIT2

#define TI_CC_GDO0_PxOUT P2OUT // cc2500 GDO0 pin
#define TI_CC_GDO0_PxIN P2IN
#define TI_CC_GDO0_PxDIR P2DIR
#define TI_CC_GDO0_PxIE P2IE
#define TI_CC_GDO0_PxIES P2IES
#define TI_CC_GDO0_PxIFG P2IFG
#define TI_CC_GDO0_PIN BIT1

#define TI_CC_GDO1_PxOUT P3OUT // cc2500 GDO1 pin
#define TI_CC_GDO1_PxIN P3IN
#define TI_CC_GDO1_PxDIR P3DIR
#define TI_CC_GDO1_PIN 0x04

#define TI_CC_GDO2_PxOUT P2OUT // cc2500 GDO2 pin
#define TI_CC_GDO2_PxIN P2IN
#define TI_CC_GDO2_PxDIR P2DIR
#define TI_CC_GDO2_PIN BIT0

#define TI_CC_CSx_PxOUT P3OUT // SPI Chip Select pin (CSx)
#define TI_CC_CSx_PxDIR P3DIR
#define TI_CC_CSx_PIN BIT0

// cc2500 Configuration Register addresses
//-----
#define TI_CCxxx0_IOCFCG2 0x00 // GDO2 output pin config
#define TI_CCxxx0_IOCFCG1 0x01 // GDO1 output pin config
#define TI_CCxxx0_IOCFCG0 0x02 // GDO0 output pin config
#define TI_CCxxx0_FIFOTHR 0x03 // RX FIFO & TX FIFO thresholds
#define TI_CCxxx0_SYNC1 0x04 // Sync word, high byte
#define TI_CCxxx0_SYNC0 0x05 // Sync word, low byte
#define TI_CCxxx0_PKTLEN 0x06 // Packet length
#define TI_CCxxx0_PKTCTRL1 0x07 // Packet automation ctrl
#define TI_CCxxx0_PKTCTRL0 0x08 // Packet automation ctrl
#define TI_CCxxx0_ADDR 0x09 // Device address
#define TI_CCxxx0_CHANNR 0x0A // Channel number
#define TI_CCxxx0_FSCTRL1 0x0B // Freq synth ctrl
#define TI_CCxxx0_FSCTRL0 0x0C // Freq synth ctrl
#define TI_CCxxx0_FREQ2 0x0D // Freq ctrl word, high byte
#define TI_CCxxx0_FREQ1 0x0E // Freq ctrl word, mid byte
#define TI_CCxxx0_FREQ0 0x0F // Freq ctrl word, low byte
#define TI_CCxxx0_MDMCFG4 0x10 // Modem config
#define TI_CCxxx0_MDMCFG3 0x11 // Modem config
#define TI_CCxxx0_MDMCFG2 0x12 // Modem config
#define TI_CCxxx0_MDMCFG1 0x13 // Modem config
#define TI_CCxxx0_MDMCFG0 0x14 // Modem config
#define TI_CCxxx0_DEVIATN 0x15 // Modem deviation setting
#define TI_CCxxx0_MCSM2 0x16 // Main Radio ctrl state mach config
#define TI_CCxxx0_MCSM1 0x17 // Main Radio ctrl state mach config
#define TI_CCxxx0_MCSM0 0x18 // Main Radio ctrl state mach config
#define TI_CCxxx0_FOCCFG 0x19 // Freq Offset Compens. config
#define TI_CCxxx0_BSCFG 0x1A // Bit synch config
#define TI_CCxxx0_AGCCTRL2 0x1B // AGC ctrl
#define TI_CCxxx0_AGCCTRL1 0x1C // AGC ctrl
#define TI_CCxxx0_AGCCTRL0 0x1D // AGC ctrl

```

```

#define TI_CCxxx0_WOREVT1 0x1E // High byte Event 0 timeout
#define TI_CCxxx0_WOREVT0 0x1F // Low byte Event 0 timeout
#define TI_CCxxx0_WORCTRL 0x20 // Wake On Radio ctrl
#define TI_CCxxx0_FREND1 0x21 // Front end RX config
#define TI_CCxxx0_FREND0 0x22 // Front end TX config
#define TI_CCxxx0_FSCAL3 0x23 // Freq synth cal.
#define TI_CCxxx0_FSCAL2 0x24 // Freq synth cal.
#define TI_CCxxx0_FSCAL1 0x25 // Freq synth cal.
#define TI_CCxxx0_FSCAL0 0x26 // Freq synth cal.
#define TI_CCxxx0_RCCTRL1 0x27 // RC oscillator config
#define TI_CCxxx0_RCCTRL0 0x28 // RC oscillator config
#define TI_CCxxx0_FSTEST 0x29 // Freq synth cal ctrl
#define TI_CCxxx0_PTEST 0x2A // Production test
#define TI_CCxxx0_AGCTEST 0x2B // AGC test
#define TI_CCxxx0_TEST2 0x2C // Various test settings
#define TI_CCxxx0_TEST1 0x2D // Various test settings
#define TI_CCxxx0_TEST0 0x2E // Various test settings

// cc2500 Strobe commands
//-----
#define TI_CCxxx0_SRES 0x30 // Reset chip.
#define TI_CCxxx0_SFSTXON 0x31 // Enab/cal freq synth
#define TI_CCxxx0_SXOFF 0x32 // Turn off crystal oscillator.
#define TI_CCxxx0_SCAL 0x33 // Cal freq synth & disable
#define TI_CCxxx0_SRX 0x34 // Enable RX.
#define TI_CCxxx0_STX 0x35 // Enable TX.
#define TI_CCxxx0_SIDLE 0x36 // Exit RX / TX
#define TI_CCxxx0_SAFC 0x37 // AFC adjustment of freq synth
#define TI_CCxxx0_SWOR 0x38 // Start auto RX poll seq.
#define TI_CCxxx0_SPWD 0x39 // Enter pwr down mode when CSn goes hi
#define TI_CCxxx0_SFRX 0x3A // Flush RX FIFO buffer.
#define TI_CCxxx0_SFTX 0x3B // Flush TX FIFO buffer.
#define TI_CCxxx0_SWORRST 0x3C // Reset real time clock.
#define TI_CCxxx0_SNOP 0x3D // No operation.

// cc2500 Status register addresses
//-----
#define TI_CCxxx0_PARTNUM 0x30 // Part number
#define TI_CCxxx0_VERSION 0x31 // Current version number
#define TI_CCxxx0_FREQEST 0x32 // Freq offset estimate
#define TI_CCxxx0_LQI 0x33 // Demod. est. for link quality
#define TI_CCxxx0_RSSI 0x34 // Rx signal strength indic.
#define TI_CCxxx0_MARCSTATE 0x35 // controlstate mach state
#define TI_CCxxx0_WORTIME1 0x36 // High byte of WOR timer
#define TI_CCxxx0_WORTIME0 0x37 // Low byte of WOR timer
#define TI_CCxxx0_PKTSTATUS 0x38 // Curr. GDOx status & packet status
#define TI_CCxxx0_VCO_VC_DAC 0x39 // Curr. setting from PLL cal module
#define TI_CCxxx0_TXBYTES 0x3A // Underflow & # of bytes in TXFIFO
#define TI_CCxxx0_RXBYTES 0x3B // Overflow & # of bytes in RXFIFO
#define TI_CCxxx0_NUM_RXBYTES 0x7F // Mask "# of bytes" in RXBYTES

// Other memory addresses
//-----
#define TI_CCxxx0_PATABLE 0x3E
#define TI_CCxxx0_TXFIFO 0x3F
#define TI_CCxxx0_RXFIFO 0x3F

```

```

// Masks for appended status bytes
//-----
#define TI_CCxxx0_RSSI_RX 0x00 // Position of RSSI byte
#define TI_CCxxx0_LQI_RX 0x01 // Position of LQI byte
#define TI_CCxxx0_CRC_OK 0x80 // Mask "CRC_OK" bit within LQI byte

// Definitions to support burst/single access:
//-----
#define TI_CCxxx0_WRITE_BURST 0x40
#define TI_CCxxx0_READ_SINGLE 0x80
#define TI_CCxxx0_READ_BURST 0xC0

// Function definition prototypes
//-----
void TI_CC_SPISetup(void);
void TI_CC_PowerupResetCCxxxx(void);
void TI_CC_SPIWriteReg(char, char);
void TI_CC_SPIWriteBurstReg(char, char*, char);
char TI_CC_SPIReadReg(char);
void TI_CC_SPIReadBurstReg(char, char *, char);
char TI_CC_SPIReadStatus(char);
void TI_CC_SPIStrobe(char);
void TI_CC_Wait(unsigned int);
void writeRFSettings(void);
void RFSendPacket(char *, char);
char RFReceivePacket(char *, char *, char *);

```


B.3.1 Block Diagram

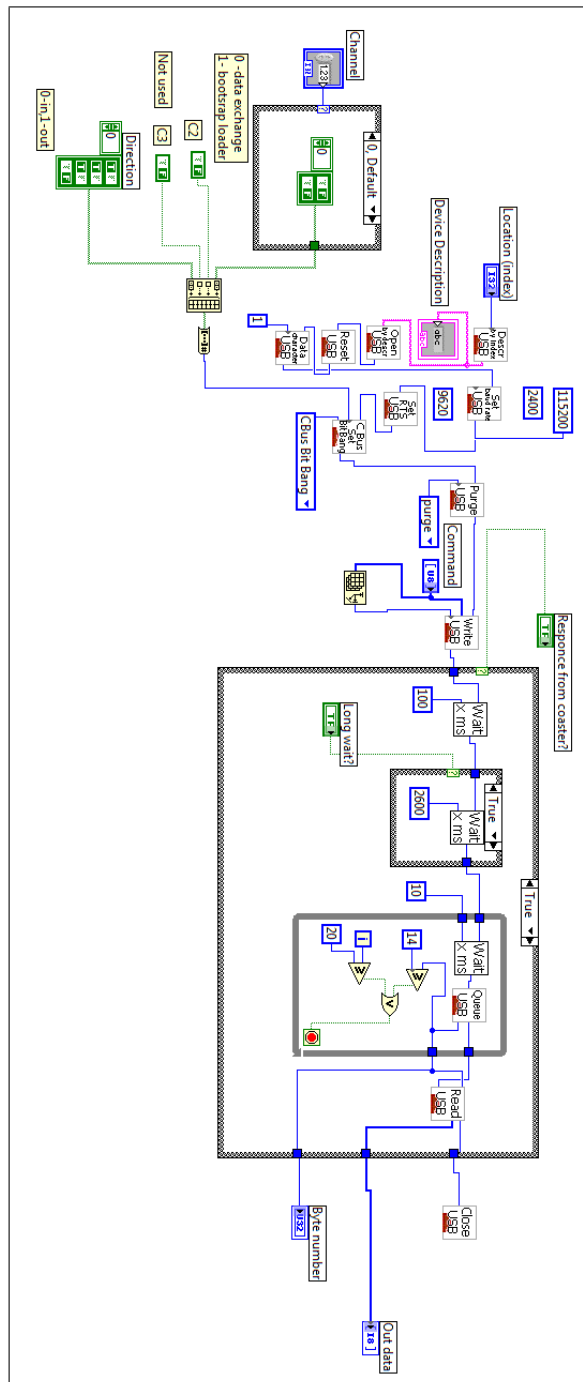


Figure B.1: LabVIEW block diagram that acquires data via USB.

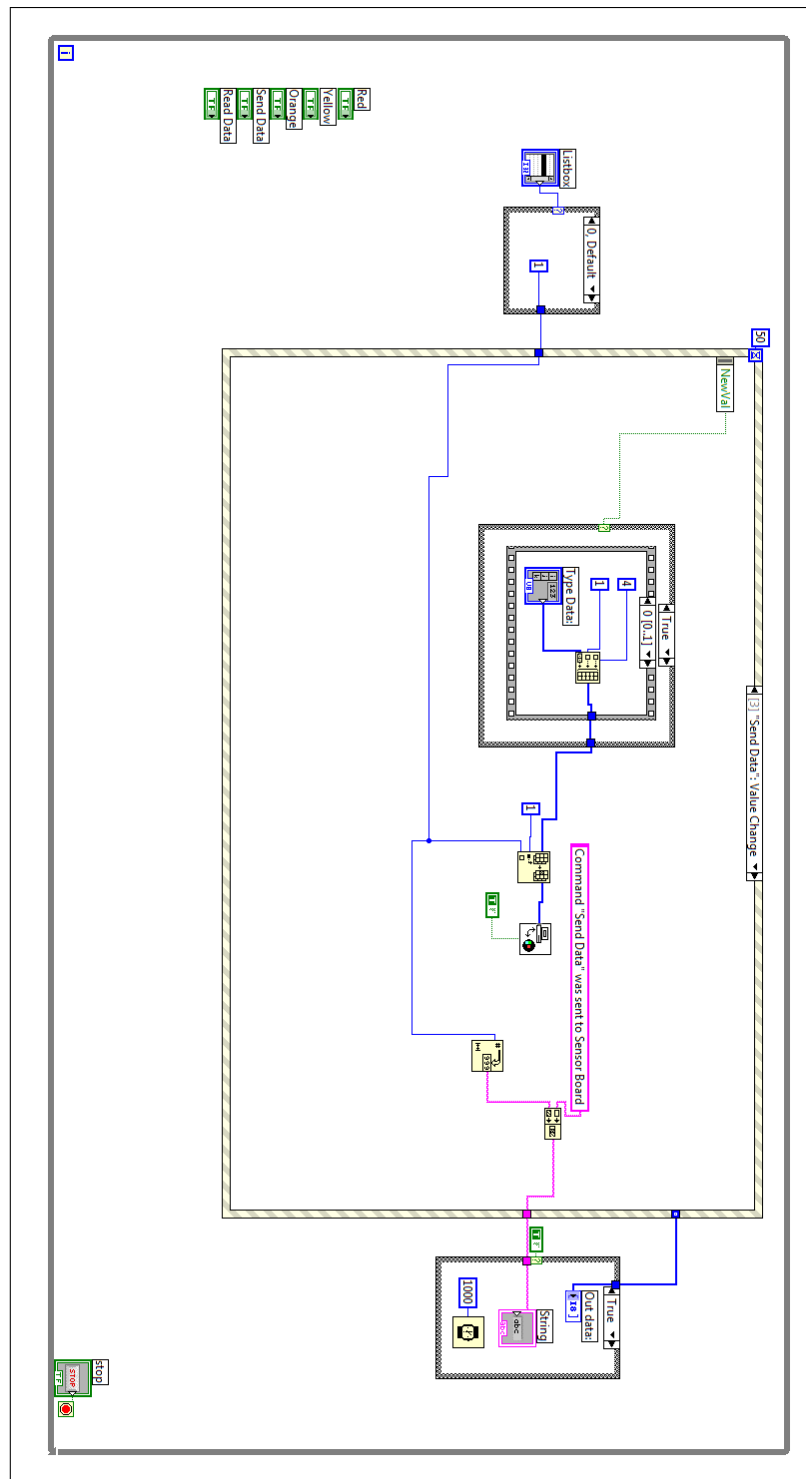


Figure B.2: LabVIEW block diagram that interfaces the Base Station with the Personal Computer.

References

- [1] S. Farahani. *ZigBee Wireless Networks and Transceivers*. Newnes/Elsevier, 2008.
- [2] Jianping Song, Song Han, Aloysius K. Mok, Deji Chen, Mike Lucas, Mark Nixon, and Wally Pratt. Wireless: Applying wireless technology in real-time industrial process control. Available on, <http://engr.uconn.edu/~song/paper/rtas08.pdf>, YEAR=2008.
- [3] Larry Friedman. SimpliciTI: Simple modular rf network specification. Technical report. Texas Instruments, Inc. Available on, <http://vip.gatech.edu/wiki/images/a/ad/SimpliciTI%2BSpecification.pdf>, last visited on February 2014.
- [4] Konstantin Mikhaylov, Nikolaos Plevritakis, and Jouni Tervonen. Performance analysis and comparison of bluetooth low energy with ieee 802.15.4 and simpliciTI. 2013. Available on, <http://www.mdpi.com/2224-2708/2/3/589/pdf>, MONTH = August.
- [5] L. Skrzypczak, D. Grimaldi, and R. Rak. Basic characteristics of zigbee and simpliciTI modules to use in measurement systems. Available on, http://www.researchgate.net/publication/228901214_Basic_Characteristics_of_ZigBee_and_SimpliciTI_Modules_to_use_in_Measurement_Systems/file/504635239699c2ccb8.pdf, YEAR=2009, MONTH = November.
- [6] Texas Instruments. *CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver*. Available on, <http://www.ti.com/lit/ds/symlink/cc2500.pdf>.
- [7] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. September 2002. Available on, <http://www.cs.berkeley.edu/~culler/papers/wsna02.pdf>, last visited on January 2014.
- [8] Cell culture basics. Invitrogen, GIBCO by life technologies. Available on, <http://www.vanderbilt.edu/viibre/CellCultureBasicsEU.pdf>, last visited on January 2014.
- [9] What is a wireless sensor network?, May 2012. National Instruments. Available on, <http://www.ni.com/white-paper/7142/en/>, last visited on January 2014.
- [10] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. WILEY, 2007.
- [11] Technologies for bioprocess ph measurement. Sensorin. Available on, <http://www.sensorin.com/upload/whitepaper10.pdf>, last visited on January 2014.
- [12] Peter Harms, Yordan Kostov, and Govind Rao. Bioprocess monitoring. 2002.

- [13] How the ph sensor works @ONLINE. Available on, <http://www.all-about-ph.com/ph-sensor.html>, last visited on January 2014.
- [14] How the ph meter works @ONLINE. Available on, <http://www.all-about-ph.com/ph-meter.html>, last visited on January 2014.
- [15] Robust nonglass ph electrode @ONLINE. Available on, <http://www.all-about-ph.com/nonglass-ph-electrode.html>, last visited on January 2014.
- [16] Optical ph sensors @ONLINE. Available on, <http://www.all-about-ph.com/optical-ph-sensors.html>, last visited on January 2014.
- [17] Mariam Naciri, Darrin Kuystermans, and Mohamed Al-Rubeai. Monitoring ph and dissolved oxygen in mammalian cell culture using optical sensors. September 2008.
- [18] Wikipedia. Oxygen sensor — Wikipedia, the free encyclopedia @ONLINE. Available on, http://en.wikipedia.org/wiki/Oxygen_sensor, last visited on January 2014.
- [19] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.
- [20] G.Z. Yang. *Body Sensor Networks*. Springer, 2006.
- [21] Wikipedia. Wireless sensor network — Wikipedia, the free encyclopedia @ONLINE. Available on, http://en.wikipedia.org/wiki/Wireless_sensor_network, last visited on February 2014.
- [22] Self-organizing networks: Wireless topologies for in-plant applications. Technical report. EMERSON Process Management. Available on, <http://www2.emersonprocess.com/siteadmincenter/PM%20Rosemount%20Documents/00840-0200-4180.pdf>, last visited on January 2014.
- [23] Wikipedia. Ieee 802 — Wikipedia, the free encyclopedia @ONLINE. Available on, http://en.wikipedia.org/wiki/IEEE_802, last visited on January 2014.
- [24] Niclas Lindblom. Small, smaller, smallest—rf communication protocols for low-power wireless systems. Technical report. IAR Systems. Available on, http://www.iar.com/Global/Resources/Developers_Toolbox/RTOS_and_Middleware/Small,%20smaller,%20smallest%E2%80%94rf%20communication%20protocols%20for%20low-power%20wireless%20systems.pdf, last visited on February 2014.
- [25] Klaus Gravgol, Jan Haase, and Christoph Grimm. Choosing the best wireless protocol for typical applications.
- [26] Wirelesshart - how it works @ONLINE. Available on, http://www.hartcomm.org/protocol/wihart/wireless_how_it_works.html, last visited on February 2014.
- [27] Joe Decuir. Bluetooth 4.0: Low energy. Technical report. CSR plc. Available on, <http://chapters.comsoc.org/vancouver/BTLER3.pdf>, last visited on April 2014.
- [28] ez430-rf2500 development tool - user's guide. Texas Instruments, Inc. Available on, <http://www.ti.com/lit/ug/slau227e/slau227e.pdf>, last visited on February 2014.

- [29] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. 2004. Available on, <http://www.eecs.harvard.edu/~mdw/papers/volcano-ewsn05.pdf>, last visited on February 2014.
- [30] Mani Srivastava, Richard Muntz, and Miodrag Potkonjak. Smart kindergarten: Sensor-based wireless networks for smart developmental problem-solving environments. 2001. Available on, http://www.cs.ucf.edu/~turgut/COURSES/EEL5937_SensorNet_Spr04/srivastava01smart.pdf, last visited on February 2014.
- [31] Insite IG, Inc. Available on, <http://www.insiteig.com/>, last visited on February 2014.
- [32] Texas Instruments. *MIXED SIGNAL MICROCONTROLLER*, July 2006. Revised August 2012. Available on, <http://www.ti.com/lit/ds/symlink/msp430f2272.pdf>.
- [33] Texas Instruments. *16-BIT, 500-KSPS, SERIAL INTERFACE MICROPPOWER, MINIATURE, SAR ANALOG-TO-DIGITAL CONVERTER*, May 2008. Available on, <http://www.ti.com.cn/general/cn/docs/lit/getliterature.tsp?genericPartNumber=ads8318&fileType=pdf>.
- [34] Texas Instruments. *MIXED SIGNAL MICROCONTROLLER*, June 2007. Revised December 2012. Available on, <http://www.ti.com/lit/ds/symlink/msp430f2471.pdf>.
- [35] FTDI Chip. *Future Technology Devices International Ltd. FT232R USB UART IC*, 2010. Available on, http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf.
- [36] Gustavo Litovsky. Beginning microcontrollers with the msp430 tutorial. Technical report. Version 0.4. Available on, http://www.glitovsky.com/Tutorialv0_4.pdf, last visited on June 2014.