Design of a GUI based on the implemented IEEE1451 standard in a Wireless Sensor Network

Bachelor Thesis

Xavier Ardanuy

#361546

24.07.2014

Supervisor: Pr. Dr.-Ing C. Gühmann

MsC. Yi Huang

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Energie- und Automatisierungstechnik
Fachgebiet Elektronische Mess- und Diagnosetechnik

“*Everything should be made as simple as possible, but no simpler.”*

Albert Einstein

Technische Universität Berlin

TU Berlin | Einsteinufer 17 | 10587 Berlin

Fakultät IV Elektrotechnik und Informatik
Institut für Energie und Automatisierungstechnik

Fachgebiet für Elektronische Mess- und Diagnosetechnik

Prof. Dr.-Ing. Clemens Gühmann

Sekretariat EN 13 Raum EN 538
Einsteinufer 17
10587 Berlin

Telefon +49 (0)30 314-29393
Telefax +49 (0)30 314-22120
clemens guehmann @ tu-berlin.de

Berlin, July 7, 2014

Bachelor thesis for Xavier Ardanuy Comas, 361546

Topic: The design of GUI based on the implemented IEEE1451 Standard in Wireless Sensor Network

Problem:

Wireless Sensor Network(WSN) is widely used in industries for monitoring and control purposes[1]. IEEE1451 is a family of high level international standard developed by the IEEE for the smart transducer, which specifies that the signal sensing and conversion functions of the smart transducer must be separated from the signal transmission and processing functions. The minmal implementation of the IEEE1451 standard based on Contiki OS[2] has been done by the chair MDT. Network Capable Application Processor(NCAP)[3] which is responsible for network communication, is implemented in Preon32 sensor node(Virtenio GmbH) and is connected to PC with USB. Several sensor nodes have been implemented as Wireless Transducer Interface Module(WTIM)[4] which store Transducer Electronic Data Sheets(TEDS). A key feature of the IEEE 1451 standard is that the Transducer Electronic Data Sheets(TEDS) is defined to store the technical data of the sensor in the non-volatile memory. Both, TEDS and the sensor data, are managed with commands, and transmitted between NCAP and WTIMs wirelessly with the defined format. However, it is difficult for the user to manage the commands without the GUI. As the result, it is necessary for users to read the TEDS and the sensor data, as well as to operate other commands with the GUI.

Task:

Considering the operability and readability, Labview could be used to design the GUI. Firstly, the communication between Labview and NCAP via USB should be explored. Secondly, all the commands in NCAP should be implemented in the design of the GUI, including the input parameters options and the output areas, such as "TIMDiscovery", "ReadTEDS", "ReadData"[3]. As the TEDS and sensor data acquired from WTIM are the raw data which is not readable, in order to display the real value, some functions should be implemented to decode the data block. Furthermore, filter option should be designed to process and to filter the signal. Users can select the proper filter and set the parameters of it for different usages.

Technische Universität Berlin

Research Steps:

1. Literature research and familar with the implemented IEEE1451 standard.
2. Learning Labview to design the GUI.
3. Designing the GUI to manage the IEEE1451 standard commands
4. Implementation of the implemented Commands.
5. Designing the filters with Labview.
6. Measurement and experiment.
7. Documentation and presentation.

Supervisor:

Prof. Dr.-Ing. Clemens Gühmann. Tel:314-29393
Email: Clemens.Guehmann@tu-berlin.de

M.Sc. Yi Huang. Tel: 314-24516
Email: yi.huang-49@mailbox.tu-berlin.de

References

[1] Robert Seng, Kang B. Lee, Eugene Y. Song: An Implementation of a Wireless Sensor Network Based on IEEE 1451.0 and 1451.5-6LoWPAN Standards

[2] Swedish Institute of Computer Science (Veranst.): The Contiki OS. http://www. contiki-os.org/

[3] IEEE Std 1451.0™-2007

[4] IEEE Std 1451.5™-2007

With kind regards

Prof.Dr.-Ing. Clemens Gühmann

TECHNISCHE UNIVERSITÄT BERLIN
Institut für Energie- und Automatisierungstechnik
Fachgebiet Elektronische Mess- und Diagnosetechnik
Einsteinufer 17 · D-10587 Berlin
Sekr. EN 13 · Tel.: (030) 314-22280

Table of Contents

GLOSSARY

ADC: Analogic to Digital Conversion
API: Aplication Programming Interface
DAC: Digital to Analogic Conversion
DSP: Digital Signal Processing
FIR: Finite Impulse Response
FXP: Fixed Point
GUI: Graphical User Interface
IEEE: Institute of Electrical and Electronics Engineers
IIR: Infinite Impulse Response
IP: Internet Protocol
I/O: Input/Output
MUX: Multiplex
NCAP: Network Capable Application Processor
OS: Operating System
TEDS: Transducer Electronic Datasheet
TIM: Transducer Interface Module
UART: Universal Asynchronous Receiver/Transmitter
VCP: Virtual Communication Port
VI: Virtual Instrument
VISA: Virtual Instrument Software Architecture
6LoWPAN: IPv6 over Low power Wireless Personal Area Networks

1 Introduction

Implementing, controlling and monitoring sensor networks, especially in wireless environments where sensor nodes are used, is becoming more and more attractive as technology advances and new applications for these networks appear. The use of a standard protocol such as IEEE1451 increases the usability and integration of the different sensors within the network. In addition, the implementation of a standard allows for the possibility to create standard systems. These standard systems can be used with no need to modify the wireless sensor network (WSN) system every time a new application wants to be implemented.

A WSN can be used in many different fields, from health care to industrial monitoring systems. For instance, they can be applied in telemedicine systems that provide remote diagnosis and treatment [1], in marine sensor networks, where accessing data in real time can become a challenge [2], or even in agriculture applications, where acquisition and processing of a collection of data coming from different sensor networks is required [3]. And these are just a few examples.

A common need in most of these systems is the existence of a user interface between the final user and the sensor network. The end user needs to have the possibility to interact with the WSN through a set of commands in order to perform actions to the sensor network such as obtain data from the transducers or read their electronic datasheets. The development of a standard communication interface in a WSN allows the engineer to abstract the sensor/network/user relation [4]. A necessary step to achieve this abstraction is the design of a user interface for the end user.

The development of a graphical user interface (GUI) increases the usability of the platform and allows for extra features that may be useful for different applications (e.g. data analysis or filter design). In addition, a user-friendly interface allows users without a deep knowledge of the structure of the network to easily interact with it with minimum amount of training required.

A GUI has to be designed with a well-defined structure and as flexible as possible to changes in the WSN in order to reduce maintenance costs and minimize the number of necessary updates to the degree possible. In this bachelor thesis a GUI to interact with the IEEE1451-based WSN implemented in the MDT Chair has been designed having these concepts in mind.

Section 2 of this document contains a brief introduction to the basic concepts to understand the structure of the WSN network our GUI will be interacting with as well as the basic concepts to understand some of the features that our GUI implements. Section 3 describes the structure and development of the GUI, its different features and some of the tests that were carried out to test the

performance of the platform. Finally, in section 4 the conclusions extracted from the development of this thesis will be presented.

2 Fundamental Concepts

2.1 The Concept of Smart Transducer

A transducer is a device that transforms energy from one form into another. Transducers allow computer systems to interact with different physical environments. Among the different transducers we can distinguish sensors and actuators. Sensors transform physical parameters into proportional electrical signals while actuators accept electrical signals and carry out a desired action. A smart transducer is the integration of an analog or digital sensor or actuator element, a processing unit, and a communication interface [5]. Fig. 1 shows the structure of a smart transducer.

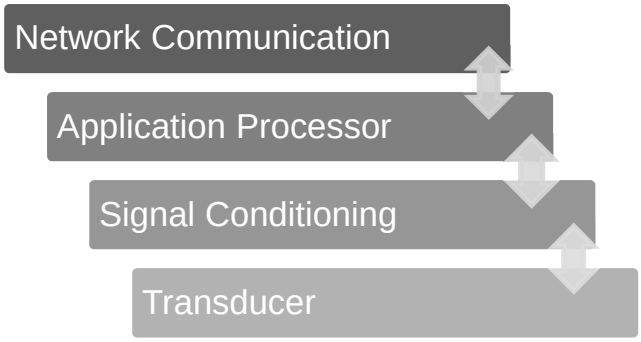

Fig. 1 Smart Transducer Structure

The IEEE1451 standard, developed by the Institute of Electrical and Electronic Engineers (IEEE) Instrumentation and Measurement Society's Technical Committee on Sensor Technology was created to satisfy the need for a set of standardized smart transducer interfaces [6]. This standard is presented in section 2.2.

2.2 Overview of the IEEE1451

The IEEE 1451 is a set of smart transducer interface standards that describes a set of open, common, network-independent communication interfaces for connecting transducers (i.e. sensors, event sensors or actuators) to microprocessors, instrumentation systems, and control/field networks [7]. One of the most important features of this standard is the definition of the Transducer Electronic Datasheet (TEDS). The TEDS electronically stores information regarding transducer's characteristics and parameters such as type of device, manufacturer, model number, serial number, calibration date, sensitivity, reference frequency, and other data. This data is stored in a memory device (usually attached to the transducer) [8].

The IEEE1451 standard family is composed of the following family of standards:

Standard	Description
1451.0	Defines Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats
1451.1	Define the Network Capable Application Processor Information Model
1451.2	Defines the Transducer to Microprocessor Communication Protocols & TEDS Formats
1451.3	Defines Digital Communication & TEDS Formats for Distributed Multidrop Systems
1451.4	Defines Mixed-Mode Communication Protocols & TEDS Formats
1451.5	Defines the Wireless Communication Protocols & Transducer Electronic Data Sheet (TEDS) Formats

Table 1 From [7] IEEE1451 standard family

As stated in the previous table, the 1451.0 describes the common functions, communication protocols, and TEDS format. This standard introduces the concept of a transducer interface module (TIM) and a network capable application processor (NCAP) connected to each other by a media specified by some other standard of the family (IEEE1451.X) [9]. The TIM is in charge of the interface, signal conditioning ADC or DAC and, usually, of containing the transducers, the number of which can go from a single one to many. On the other hand, the NCAP acts as a liaison between the different TIMs of the transducer network and control each TIM by means of an interface medium, which allows for performing different actions, such as read and write TEDS of a certain transducer and read data from a certain sensor, for instance. As it will be described in section 2.3, the communication between the different TIMs and the NCAP in our WSN is provided by the IEEE1451.5 standard, which describes the wireless communication protocols used by our WSN.

To be able to develop actions such as read TEDS from a transducer; this standard defines an Application Programming Interface (API) that provides communications between users' network and the IEEE1451.0 layer of the NCAP. Another API is also provided between IEEE1451.0 and the other standard (in our case, IEEE1451.5) used to perform the communication between the NCAP and the TIMs. Please refer to section 2.3 for an overview of the implementation of these standards in the WSN used in this bachelor thesis.

The image below shows the structure of the relationship between the different IEEE1451 standards family members

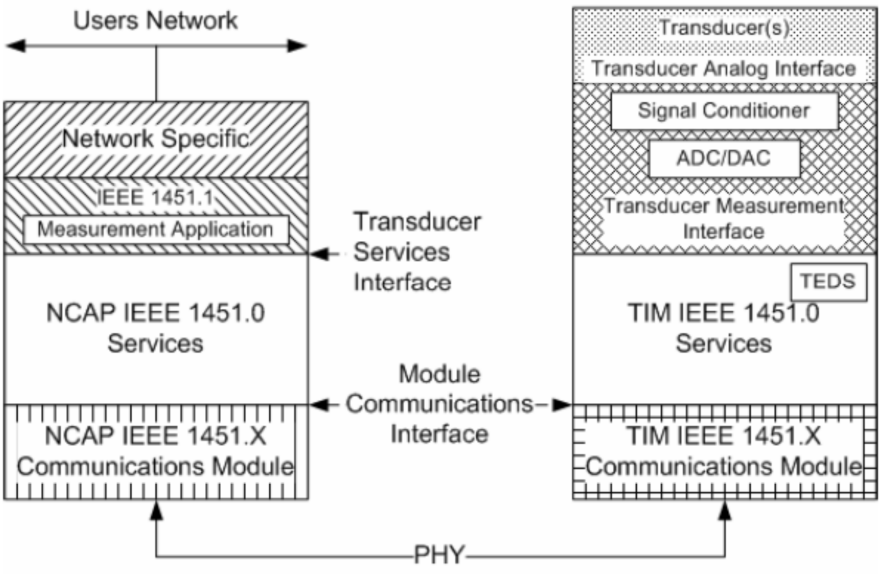

Fig. 2 From [9] Reference model for the IEEE1451 standard

2.2.1 Transducer Electronic Data Sheet (TEDS)

The TEDS are blocks of information that describe a TransducerChannel, generally stored in non-volatile memory within a TIM. The concept of TransducerChannel refers to a transducer and all of the signal conditioning and conversion components associated with that transducer. Normally the TEDS of a certain transducerChannel are not changed once the user or the manufacturer establishes them [9].

We can differentiate between three different transducerChannels:

- Sensors: a sensor measures some physical parameter and returns the values captured in a digital format.
- Event Sensors: Unlike the sensor, an event sensor determines when a change of state occurs. This change of state could be, for example, an analog signal crossing a certain threshold.
- Actuators: An actuator acts in the opposite way of a sensor; it performs an action upon the receipt of a trigger.

The IEEE1451 standard defines four TEDS as mandatory for all TIMs, while the others are optional [9]. Table 2 lists the mandatory TEDS and gives a short description of each one of them. The mandatory TEDS are the only ones implemented in the sensor nodes used in this bachelor thesis.

Type	Description
Meta-TEDS	Contains the necessary information to gain access to any TransducerChannel as well as information common to all TransducerChannels
TransducerChannel TEDS	Contains the information concerning a certain TransducerChannel to enable its proper operation
User's Transducer Name TEDS	Contains a place to store the name by which the system or the end user will know a certain transducer
PHY TEDS	Contains the information needed to gain access to any channel, plus the information common to all channels. Details of this TEDS will depend on the standard used for communication (In this thesis, it is the IEEE1451.5)

Table 2 From [9] Required TEDS

In addition to the previously presented TEDS, the standard describes the following TEDS as optional:

- Calibration TEDS
- Frequency Respons
- Transfer Function TEDS
- Commands TEDS
- Identification TEDS
- Geographic location TEDS
- Units extension TEDS
- Manufacturer defined TEDS

More information on the TEDS specifications can be found in the IEEE1451.0 standard document [9].

2.3 MDT Smart Transducer Library (MSTL)

The MDT Chair of the TU Berlin has developed the MDT Smart Transducer Library (MSTL), a universal interface to a wireless sensor nodes network. This interface is based on the previously described IEEE1451 standard. This interface is still under development and, therefore, still not totally compliant with the standard.

The MDT WSN used in this bachelor thesis is composed of different sensor nodes implemented in Preon32 devices (please refer to section 2.4.1 for more details on the hardware), which make use of Contiki OS, an operating system used for low-power Internet communication (refer to section 2.4.2 for more details on the Contiki OS). Therefore, both TIM and NCAP are implemented in the same hardware and OS and it is only the application loaded to them what differentiates them.

An overview of the structure of the WSN used in this bachelor thesis is presented below:

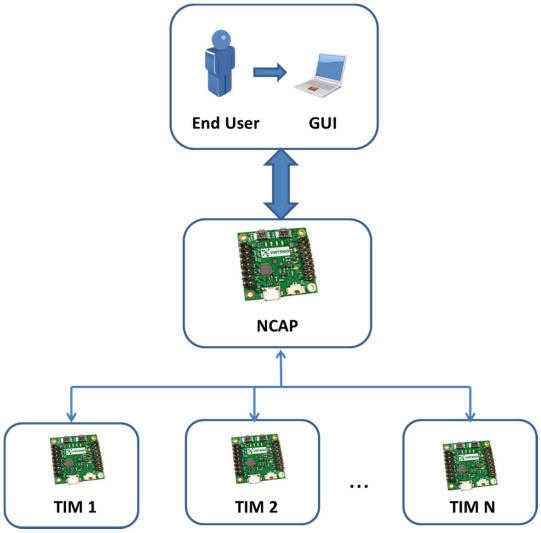

Fig. 3 Overview of the Structure of the Wireless Sensor Network (WSN)

The end user makes use of the user interface using a computer to communicate with the IEEE1451.0 layer implemented in the NCAP. This interface is managed by a serial shell. Contiki provides with a command-line shell that allows users to send commands to a contiki system from a computer. In the MSTL, a special set of commands has been included to allow for interaction with the IEEE1451.0 layer of the NCAP. This *shell-ncap* has been developed following the NCAP HTTP-API described in section 12 of the IEEE1451.0 standard [9]. The NCAP acts as a base station and interacts with the different TIMs present in the sensor network. The wireless communication protocol between the NCAP and the different TIM is defined by the IEEE1451.5 standard. The physical communication and the access control are implemented following the 802.15.4 wireless communication standard. As a final step of the chain, the TIM executes the commands sent by the NCAP. The TIM contains the different transducers, its corresponding TEDS stored in a flash memory and it is charge of the ADC and signal processing of the data received from the sensors.

The following scheme shows the data flow in our system when it comes to acquire data from a certain sensor to our computer.

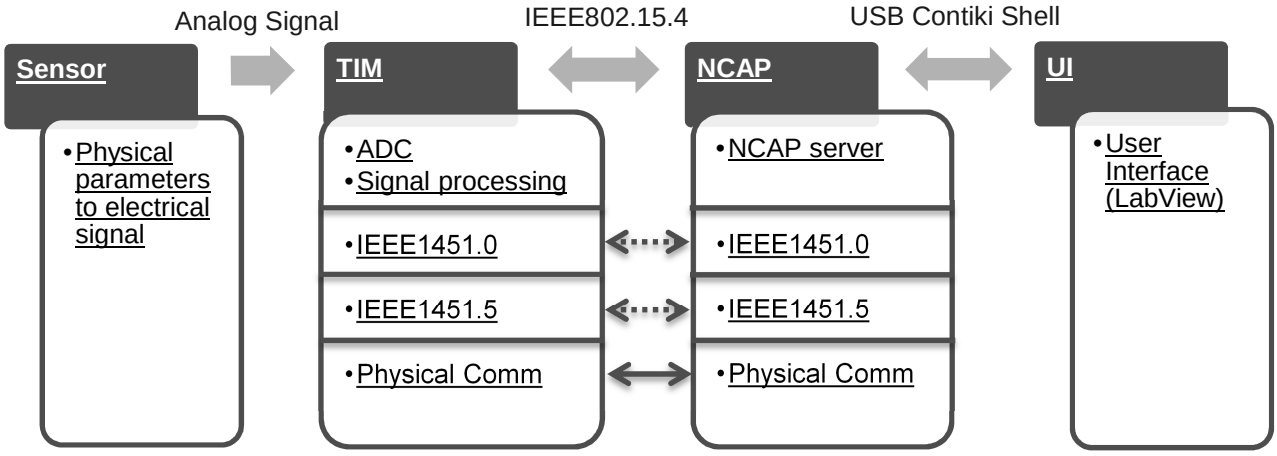

Fig. 4 Data Acquisition Dataflow

2.4 Hardware and Software Tools

This section presents the hardware and software tools used in this bachelor thesis. Special emphasis will be given to the LabVIEW software, which has been used to design the GUI later described in section 2.5

2.4.1 Virtenio Preon32

The sensor nodes of our sensor network are implemented in Virtenio Preon32 platforms. The Preon32 contains a 32 bit microcontroller (STM32-microprocessor), an IEEE802.15.4 compliant RF transceiver and an 8Mb serial flash memory for measurement and configuration of data [10].

The Preon32 sensor nodes used in this bachelor thesis are placed in an application board (Preon32-shuttle), which provides additional interfaces appropriate for prototyping purposes such as LEDs or buttons. In addition, this board provides with a USB to UART Bridge, which allows for an easy data transfer to other devices, such as a computer, as will be done in this project.

Fig. 5 Preon32

Fig. 6 Preon32-Shuttle

2.4.2 Contiki OS

The sensor nodes used in this bachelor thesis implement the Contiki OS. This operating system is designed to connect small, low power microcontrollers to the internet. Contiki provides with different characteristics that makes it suitable for small resource-limited sensor nodes in a wireless network [11]. These characteristics are, for example, the fully support of many wireless standards, among which we can find the 6lowpan (IPv6 over Low power Wireless Personal Area Networks), which is used in our sensor network over the IEEE802.15.4 layer. Another useful feature of Contiki is the Contiki-shell, which provides with a set of commands to interact with Contiki systems. Furthermore, Contiki has an active community and it is open source software, being the code completely available to everybody.

2.4.3 LabVIEW

LabVIEW is a programming language developed by National Instruments Corporation. LabVIEW is an acronym for "Laboratory Virtual Instrument Engineering Workbench". It is a graphical language where no written code is used, but a block diagram programming structure, where developers can easily manage the data flow between the different parts of the program. LabVIEW is specially designed for measurement and control system, providing a wide range of tools for data acquisition, data processing and communication to different kind of devices. [12]

LabVIEW provides developers with two windows. The front panel, which allows developers to design the graphical interface, and the block diagram, which contains the code run by the application. The blocks contained in the block diagram are called Virtual Instruments (VI). Every application in labVIEW is a VI and all advanced VIs contain other sub-VIs inside their block diagrams executing specific routines.

A key feature of LabVIEW is the possibility to create user-friendly interfaces that allow for the development of applications that can be used by users without special expertise in a certain matter.

These user friendly interfaces can be compiled in an executable file making it possible to run the application in different computers just by opening a simple application in their desktop. In addition, a setup file can also be created to allow users to install the application in their computer with no need to have LabVIEW runtime engine previously installed, which also avoids possible version incompatibilities.

In this bachelor thesis, LabVIEW 2013 has been used to develop the different parts of the GUI. In addition, the Digital Filter Toolkit, a separate module of LabVIEW, has also been used to develop the parts of the GUI regarding filter design and analysis. Section 3 of this thesis describes the development of the GUI using LabVIEW.

2.5 Filter Design

The user interface developed in this bachelor thesis allows the user for the design of discrete time filters in an interactive way. The user can use different discrete time filter design techniques and analyse the filter obtained in order to reach a good approximation of the desired response. The coefficients of the designed filter can afterwards be quantized and transmitted to a TIM of the sensor network, which will apply the filter to the specified sensor signal.

In this subsection we are going to analyse the different filter design techniques implemented in the user interface and the concept of fixed point coefficients.

2.5.1 Discrete Filter design techniques

When designing filters we try to obtain the best approximation that we can get of the desired ideal filter. To do so we determine the filter specifications in the frequency domain and obtain the impulse response coefficients of the filter in the time domain using an appropriate algorithm. Usually these specifications are given in the analog domain. However, since we will always know the sampling frequency it will be direct for our application to convert the specifications from the analog to the discrete domain.

There are two main approaches when designing digital filters. The first one designs the filter in the analog domain and then converts it to its discrete equivalent. With this method we can obtain infinite impulse response filters, or IIR filters. The other method consists on designing filters directly in the discrete domain. In this case we obtain finite impulse response filters, or FIR filters [13]

In our user interface, however, we will only be working with FIR filters. IIR filters might sometimes be easy to implement and they give good results, since the art of analog filters is very well known. Nevertheless, several reasons, such as stability, make it more suitable to implement FIR filters in our system. More details on the comparison between IIR and FIR filters can be found in section 2.5.1.1

2.5.1.1 IIR vs FIR Filters

In the time domain, the following convolution of N+ 1 coefficients defines the filtering process of a FIR filter:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] \quad (1)$$

Where y[n] is the output of the filter, and N is the filter order. It can be easily seen that if a single non-zero value is present at the input of the filter but all subsequent input samples are zero, the output of the filter in (1) becomes zero after the filter processes N+1 input data samples [13].

The stability of a filter can be analysed observing the position of the poles of the filter in the pole-zero diagram. It is known that for a filter to be stable, all poles have to be inside the unit circle. As seen in equation (1) a FIR filter has only zeroes. Therefore, we do not have to worry about the stability of the filter, since an FIR filter will always be stable

On the other hand, we have the following recursive equation that defines the filtering process of an IIR filter:

$$\sum_{i=0}^{Q} a_i y[n-i] = \sum_{k=0}^{P} b_k x[n-k] \quad (2)$$

Where P is the numerator order and Q is the denominator order. The filter operates on current input x[n] and previous input x[n–k] and current output y[n] and previous output y[n–i]. The impulse response of the filter in this case is infinite because the filter might generate non-zero outputs far into the future in response to a single non-zero input sample [13]. Another way to express the equation of an IIR filter, in its z transform function, is shown below:

$$H(z) = \frac{\sum_{k=0}^{P} b_k z^{-k}}{1 + \sum_{i=1}^{Q} a_i z^{-i}} \quad (3)$$

In this kind of filters the stability becomes an issue since we have to control the position of the poles. The design of such becomes also trickier. In addition, when transforming the floating point version of the filter to a fixed point, the stability has to be analysed again, since the filter can become unstable due to the quantification of the coefficients.

The following table summarises the main differences between FIR and IIR filters:

Feature	FIR Filter	IIR Filter
Closed Formula	Not possible	Possible
Stability	Always stable	Conditionally stable
Fixed Point version	Easy to implement	Complicated to implement
Exact Linear Phase?	Always linear	Not possible
Computational effort	More computations	Fewer computations

Table 3 IIR vs FIR Filter Comparison

The main reasons that make it more suitable to implement FIR filters are the stability, the linear phase and the easiness to implement fixed point versions of the filter. A fixed point version of the designed filter will be implemented since a floating point would require a much greater effort to the sensor node than a fixed point one, which makes it easier and fairly good results can be obtained.

2.5.1.2 Filter Design Techniques implemented in the GUI

LabVIEW provides developers the possibility to work with many different kinds of filters. We can divide them into two types: the ones designed using the classical filter design VI and the ones designed with advanced and special filter VIs. Since we are only willing to implement FIR filters, we will only talk about FIR filters in both the classic and the advanced design approach.

In our GUI, the user is able to design filters using both the classical filter design technique as well as advanced techniques. The Classical Filter Design VI provides with an easy tool to design different types of filters and for many applications, the results obtained are sufficient.

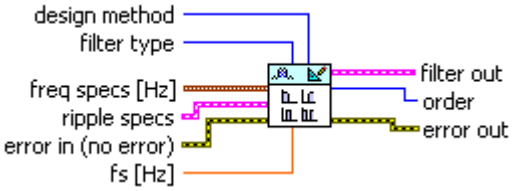

Fig. 7 Classical Filter Design VI

This VI provides with the following FIR filter design methods:

- Kaiser Window
- Dolph Chebyshev Window
- Equi-Ripple FIR

In addition to the presented VIs, our GUI will provide with more advanced filter design VIs. These VIs might not be as easy to use as the previous one since the user requires a certain level of knowledge regarding FIR filter design. These VIs are:

- Remez Filter Design
- Window filter Design
- Least pht filter Design

The Kaiser Window and the Dolph-Chebyshev Window are particular cases of the well-known window design method. The FIR Window design method is one of the simplest methods. The basic idea behind it is the application of a window to an infinitely long filter to convert it into an FIR. The Equi-Ripple, Remez and Least pth are methods that use optimization algorithms to obtain the filter coefficients and for certain applications their results are optimum and worth using. Details on these techniques and their implemetation in LabVIEW can be found in LabVIEW Digital Filter Toolkit Manual [14].

2.5.2 Fixed Point Filters

In our GUI we will be implementing fixed point filter design. Fixed point filters are a simplified form of a floating point filter where all coefficients are quantized and represented with the same number of bits. Fixed point filters are suitable for applications such as ours, where filtering functions have to be carried out in limited embedded systems where floating point operations are relatively expensive in terms of computation effort. The use of fixed point filters, however, might compromise the performance of our filter due to errors introduced during the quantization and filtering processes such as overflow saturation, arithmetic rounding or coefficient rounding [15]. Therefore, the design of fixed point filters will require an extra analysis and simulation, based on a try and error procedure, until we obtain the desired result.

Coefficients are designed in the GUI in floating point format, transformed to fixed point numbers (in QFormat) and sent to the TIM. The TIM is then in charge of filtering the signal using the coefficients sent. The TIM uses ARM filter functions of the Cortex Microcontroller Software Interface Standard (CMSIS), an independent hardware abstraction layer for the Cortex-M processor series. The functions that our TIM will be working with deal with fixed point coefficients in Q15 format. Details on the CMSIS functions can be found in the CMSIS website [16].

Qformat is usually used to determine fixed point numbers and its notation is written as Qm.n [17].

- *m* determines the numbers of bits to determine the integer part of the number
- *n* determines de number of bits used to determine the fractional part of the number

The most significant bit is designated as the sign bit, since the number is in two's complement format (See equation 4). Thus, in order to represent the whole number we will need *m + n + 1* bits. Therefore, the range of numbers that can be represented is $[2^m, 2^m - 2^{-n}]$.

In our application we will be working with Q15 format. This format is, in reality, Q0.15. No bit is used to represent the integer part since all coefficients are between ± 1. See below the structure of a Q15 filter coefficient:

BIT	15	14	13	**…**	0
VALUE	Sign (b_{15})	Q14 (b_{14})	Q13 (b_{13})	**…**	Q0 (b_0)

Table 4 Q15 Number Format

$$x = -b_{15}2^0 + b_{14}2^{-1} + b_{13}2^{-2} + \cdots + b_1\ 2^{-14} + b_0\ 2^{-15} \quad (4)$$

The range of values is $[-1, 1 - 2^{-15}]$ which is [-1, 0.9999694824]. With a Q15 structure we have, therefore, a precision of 2^{-15}, which is the difference between two consecutive fixed point Q15 numbers.

The transformation (quantization) of the floating point coefficients to Q15 values follows a simple procedure:

Let x be a floating point coefficient and x_q its Q15 version:

$$x_q = round(x \cdot 2^{15}) \quad (5)$$

The integer x_q obtained in equation (5) is the value that will be sent as the value of the filter coefficient.

3 Graphical User Interface Development

This section describes the development of the graphical user interface that allows users to interface with the sensor network using a user-friendly application running in their computer. Firstly, the interface between the GUI and the NCAP will be described. Then, the structure of the GUI will be detailed as well as the different features that it offers. Finally, a separate section will describe some of the different tests that were carried out to GUI.

3.1 Interface between the GUI and the Sensor Node

As explained in section 2.4.1, the preon32-shuttle used in this bachelor thesis has a UART to USB Bridge which allows for communication between the sensor node and the computer. Therefore, there is a need to do the inverse procedure and adapt again the USB communication to a serial port communication on the computer side. To do so, a CP210x USB to UART Bridge VCP driver provided by Virtenio was used. This driver creates a virtual communication port in the computer side allowing the communication between the end user and the sensor node to be regarded as a simple serial port communication. This implementation avoids the problem of implementing other kinds of communication, such as raw USB communication, far more complicated and prone to implementation errors.

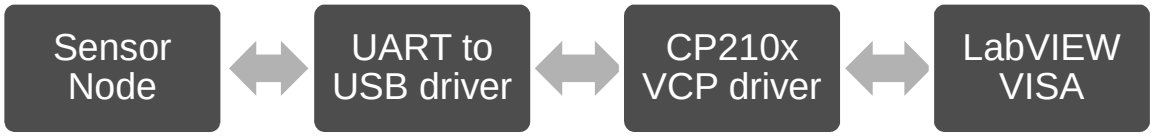

Fig. 8 Overview of the Interface between the GUI and the NCAP

LabVIEW provides with a set of functions that allow for the serial communication between labVIEW and our sensor nodes. These functions are contained in the Virtual Instrument Software Architecture (VISA), a standard for configuring, programming and troubleshooting instrumentation systems comprising different interfaces (GPIB, VXI, PXI…) [18]. For our GUI design we will only be using the VISA serial communication functions that will allow us, among other features, to open and close a connection with our sensor node or read and write commands. Both read and write functions of the VISA use strings as inputs and outputs, and since the application will be receiving binary data, we will have to transform the received string into a byte array in order to work with it, as can be seen in Fig. 10.

The commands sent by our application and, therefore, written through the VISA-Write function are shell-ncap commands. These commands, defined in the IEEE1451.0 standard provide with the interface between the IEEE1451 and the user.

3.1.1 Shell communication

The serial communication between the GUI and the NCAP is carried out through the Contiki-shell, as already mentioned. The shell commands used in this bachelor thesis are the ones that allow for the interaction with the IEEE1451.0 layer in the NCAP. These commands are:

Command	Description
TIMDiscovery	Searches TIM available in the WSN
TransducerDiscovery	Searches for transducers available in a TIM
ReadTEDS	Reads the TEDS information of a transducer channel
ReadData	Read data from a transducer
WriteData	Writes data to a transducer (e.g. filter coefficients for signal processing)
SendCommand	Executes a command to a transducerChannel

Table 5 Shell-NCAP commands

3.1.1.1 Command Format

All commands and their arguments are transmitted through the Contiki-shell as ASCII strings. In case of long arguments, such as the ones in WriteData (e.g. sample period, filter coefficients, etc.) these arguments are sent as binary data encoded in base64-encoded strings. Arguments are separated by a space character (ASCII: 32) and each command is finished with a line feed characther(ASCII: 10). In Fig. 9 the "TIMDiscovery" string command is sent to the ID Discovery function. This function sends the input string command to the NCAP, reads the response and parses the received data in order to have a list of IDs (In this case, a list of TIMs available in the network

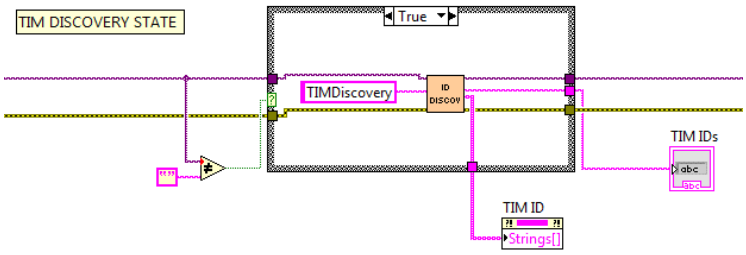

Fig. 9 *TIMDiscovery* command sent to the NCAP

3.1.1.2 Response Format

The command responses are received in binary format. The first two bytes correspond to the Error Code. The next two bytes indicate the number of bytes that follow. After these four bytes we receive the actual data. For each command, the structure of this data is different.

Error Code (uint16)	Length (uint16)	Data

Table 6 Command Response Format

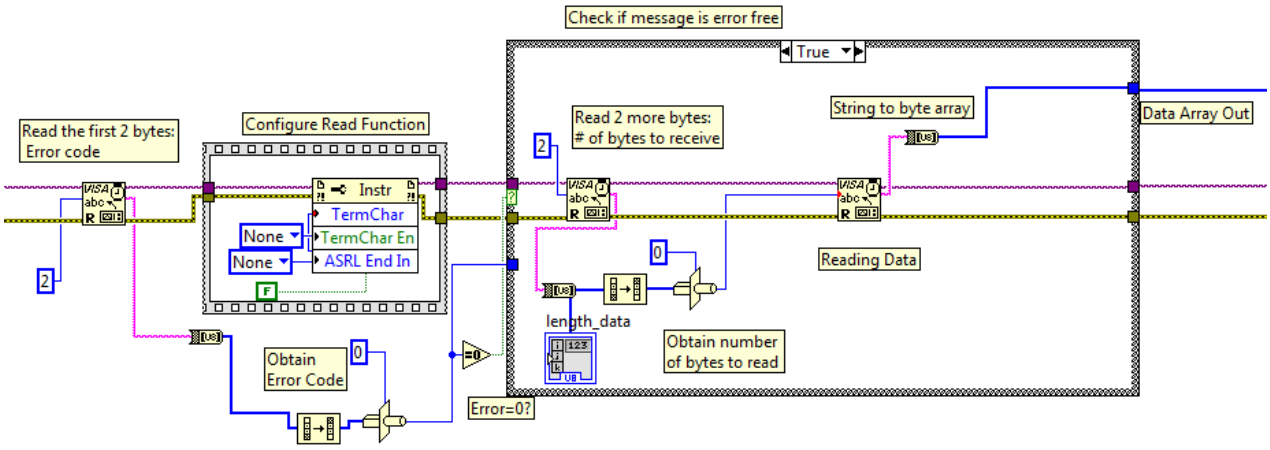

Fig. 10 Read Reply (Without Error)

The Error Code is zero in case there is no error reported by the sensor network. If there is an error, then this code has to be decoded and reported to the user. In that case, no more bytes are read and the error is handled by the function *check error* (Fig. 11). More details on the error handling by the GUI can be found in section 3.3

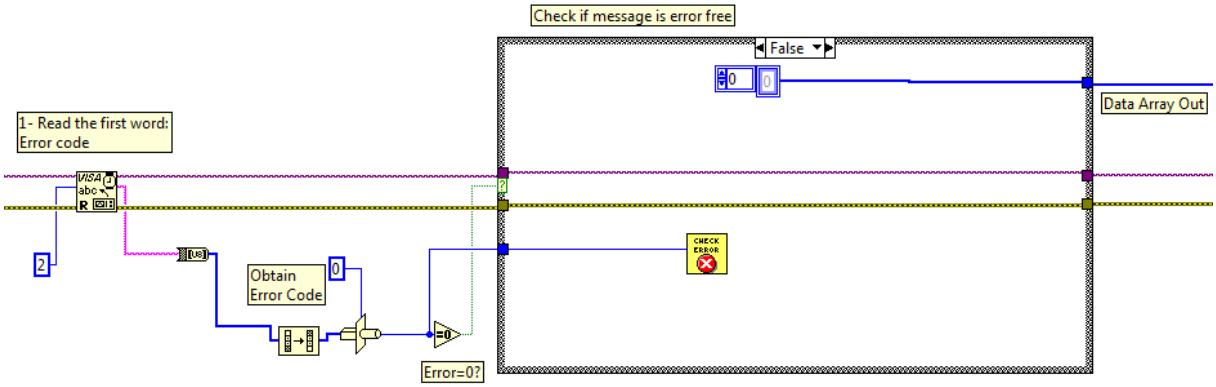

Fig. 11 Read Reply (With error)

3.2 GUI structure and features

3.2.1 GUI Design – Event Structured State Machine

From the graphical design point of view, the GUI has been designed having in mind the importance of a user-friendly graphical interface to allow users to interact with it with the least amount of training possible. In fact, only one window is used and users can navigate through different tabs as seen in Fig. 12. Through the different tabs the users can interact with the IEEE1451.0 layer in the NCAP using the different command options available. The existence of different windows for each feature would make the interface more confusing and disorganised. To the extent possible, the GUI has been designed following the user interface design rules described in [19].

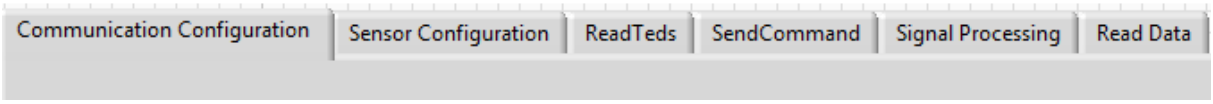

Fig. 12 Tab Control in the Front Panel

From the programming point of view, the GUI has been designed as an event structured state machine. A state machine is a common architecture in LabVIEW programs which implements an algorithm described by a "Moore machine". A Moore machine performs a specific action for each state. Each state can lead to one or multiple states or end the process flow depending on the user input or on in-state results [20].

In our user interface, user actions (events) lead to a change of state. When the GUI is initiated, the state machine moves to the default state (*IDLE* state) and stays there until an event is detected. Events are managed by an event handler and each one of these events leads to a different state. Then, each state performs its correspondent set of actions and, if there is no communication error (if there can be any), the next state is again the *IDLE* state. Therefore, most states lead to the *IDLE* state except for the *Stop* state, which shuts down the application. The state diagram implemented in our GUI can be seen in Fig. 13.

The instructions carried out in the different states require little amount of time so the idea is that most of the time the state machine stays in *IDLE* state, and when an event occurs, the instructions related to this event are executed and we return again to *IDLE* state. Since the amount of time in another state is relatively short, the possibility for an event to occur while being in a state different from *IDLE* is relatively small. In case that happened, the code execution of the interrupted state would be left unfinished, which would, most of the time, lead to an error (Please refer to section 3.3 for details on the error handling in our GUI)

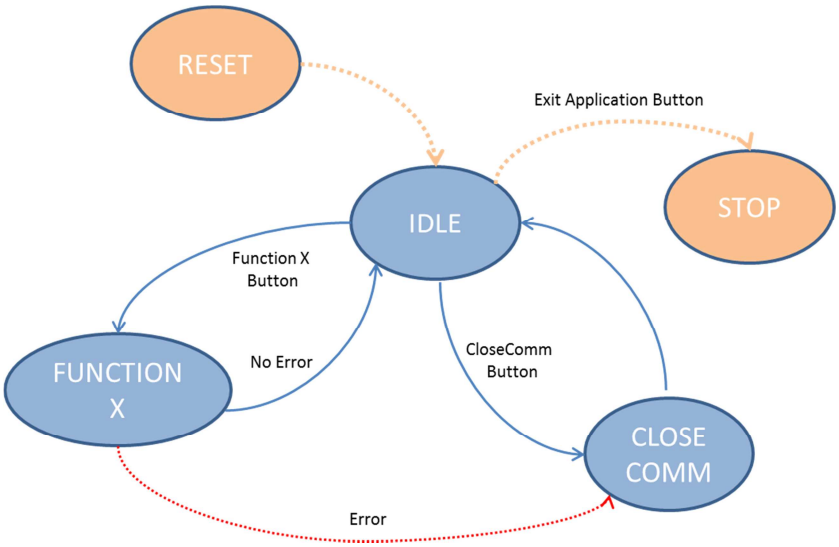

Fig. 13 State Machine Simplified Diagram

As observed in the diagram above, the change of state is carried out when the user clicks on a certain button. In our GUI, these events are handled by an *event structure* inside a *while* loop (Fig. 14), which listens to events occurred in the front panel.

When a button is pressed, the event handler executes the code related to that event. In our program this code consists of changing the value of the variable *state* (as seen in Fig. 14). When this code is executed, after a defined period of time the *event structure* moves to *Timeout*

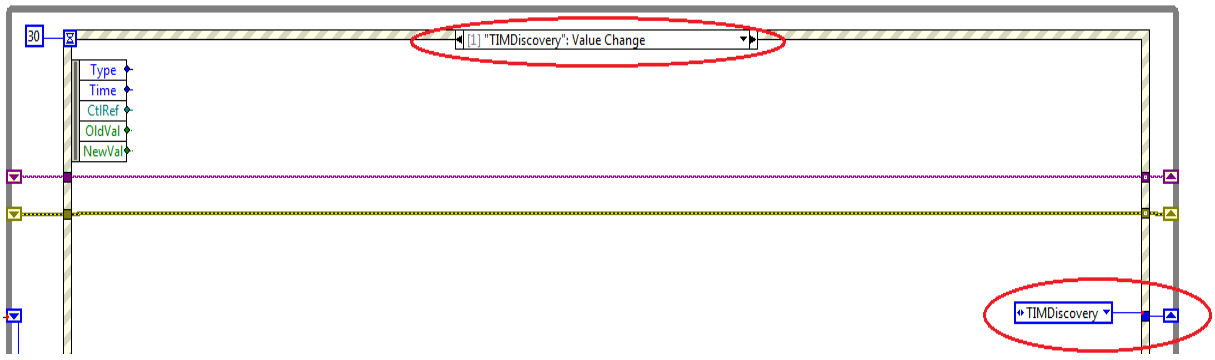

Fig. 14 Event structure inside a while loop. In this figure we are handling the *TIMDiscovery* button click. When this event occurs, the value of the variable *state* is set to TIMDiscovery (see right side of the figure)

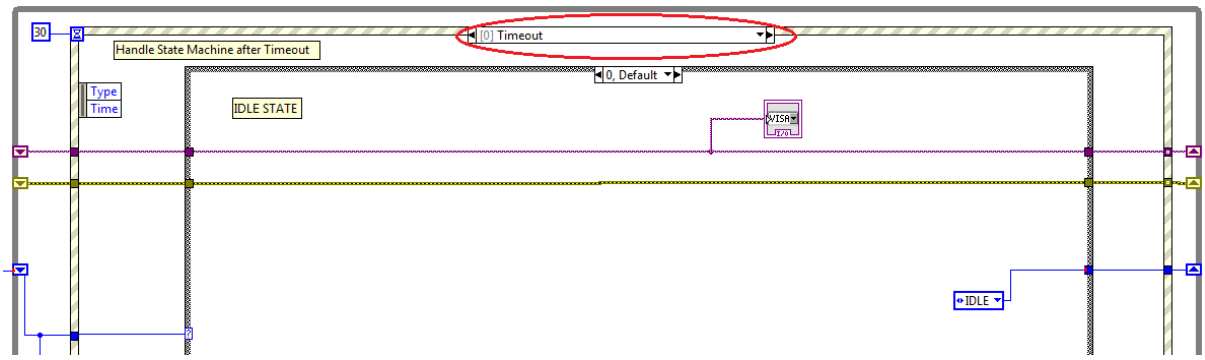

Fig. 15 Event structure handling timeout

When the event handler moves to *Timeout*, a *case structure* (Fig. 16) checks the value of the *state* variable. Depending on the value of this variable, the case structure executes different code. Once the code is executed, the variable *state* is set depending on which has to be the next state.

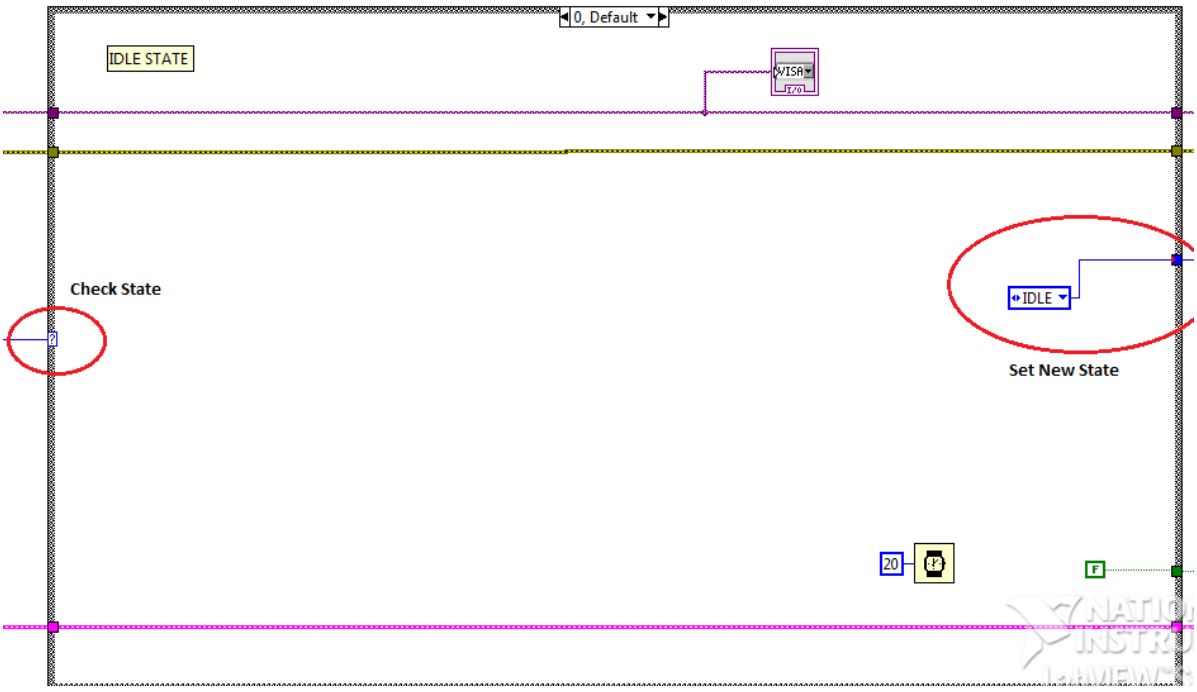

Fig. 16 *Case structure* executed after timeout.

At the left side of Fig. 16 we can see how the case structure checks the value of the current state. At the right side, a new value is set to the variable *state*. In this figure, the *case structure* handles the *IDLE* state, so no code is implemented (only a short delay) and the next state is set again to *IDLE*.

The *case structure* controls the code that has to be executed every time there is a timeout after an event occurs. See in the following subsection the different states implemented in our event structured state machine, when they occur and what is the state that follows in each case.

3.2.1.1 IDLE state

This is the default state. Once the application is started we move to this state until an event occurs. This state case does not carry out any action. Since we want to stay in *IDLE* state until an event occurs, the default output state is always *IDLE* (see Fig. 16). Therefore, the only way to move from IDLE state is an event to happen. This event is managed by the event handler, which will change the variable *state*.

3.2.1.2 Init Comm State

This state is reached when the user clicks on the *Init Comm* button situated in the Communication Configuration tab of the front panel. In this state, the application establishes a serial line communication between the GUI and the NCAP.

This state has two possible output states. If there is an error in the connection establishment, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state

3.2.1.3 Close Comm State

This state is reached when a user clicks on the *Close Comm* button situated on the Communication Configuration tab of the front panel or when a VISA error occurs in another state. This state simply closes the serial line communication and shows a message to the user if an error has been detected. This error might have occurred in some other VISA VI but it is the *VISA Close VI*, present in the *Close Comm* State, the one in charge of reporting this error.

3.2.1.4 TIMDiscovery State

This state is reached when the user clicks on the *TIMDiscovery* button situated in the Communication Configuration tab of the front panel. In this state, the application sends a *TIMDiscovery* command to the NCAP in order to receive a list of the available TIM in the WSN. For more details regarding features of the Communication Configuration tab, please refer to section 3.2.2.1

This state has two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state.

3.2.1.5 TransducerChannel Discovery State

This state is reached when the user clicks on the *TIM Selection* button situated in the Communication Configuration tab of the front panel. This state carries out the necessary actions to send a *Transducer Discovery* command to the NCAP in order to receive a list of the available transducerChannels in a TIM.

This state has two possible output states. If there is an error in a *VISA* VI, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state.

3.2.1.6 Sensor Configuration State

This state is reached when the user clicks on the *Configure Sensor* button in the Sensor Configuration tab of the front panel. In this state, the application sends the necessary data to the TIM in order to configure the sensor sampling period, number of samples and number of the ADC channel we want to read from. More details on the sensor configuration features can be found in section 3.2.2.1.1

This state has two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state

3.2.1.7 ReadTEDS state

This state is reached when the user clicks on the *ReadTEDS* button in the Read TEDS tab of the front panel. This state carries out the necessary functions to send a *ReadTEDS* query to a certain TIM in order to get the values of the TEDS of a certain transducerChannel. More details on the *ReadTEDS* feature can be found in section 3.2.2.3

This state has two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state

3.2.1.8 SendCommand State

This state is reached when the user clicks on the *SendCommand* button in the Send Command tab of the front panel. In this state, the necessary functions to send a command to a transducerChannel of a certain TIM are implemented. More information on the Send Command feature can be found in 3.2.2.4

This state has two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state

3.2.1.9 Filter Design State

This state is reached when a user clicks on the *Design Filter* button in the Signal Processing tab of the front panel. In this state, a fixed point filter is designed following the specifications introduced by the user. Once designed this filter, it can be analysed and later applied to a certain TransducerChannel. More details on the filter design feature can be found in section 3.2.2.5

This state has only a possible output state, IDLE state.

3.2.1.10 Apply Filter State

This state is reached when a user clicks on the *Apply Filter* button in the Signal Processing tab of the front panel. In this state, the designed filter in the *Filter Design* state is sent to a certain TIM so it can be applied to a signal coming from a sensor.

This state has two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state.

3.2.1.11 ReadData State

This state is reached when a user clicks on the *ReadData* button in the Read Data tab of the front panel. In this state, the necessary functions are implemented to read data from a certain transducerChannel. This data is displayed in a graph and other relevant parameters are also displayed. More details on the read data feature are presented in section 3.2.2.6

This state has also two possible output states. If there is an error in a *VISA VI*, the output *state* variable is set to *Close Comm*. however, if there is no error, the output state variable is set to *IDLE* and the state machines returns to the initial state.

3.2.1.12 Stop State

This state is reached when the exit button is pressed. When this happens, if there is any serial communication session opened, it is closed and the program exits the while loop and the application is shut down.

Since this state leads to the shutdown of the application, there is no output state.

3.2.2 GUI Features

This section describes the different features that our GUI offers the users. The user can easily interact with these features by moving through the different tabs of the front panel (Fig. 12). The subsections below describe in detail each one of these features.

3.2.2.1 Communication Configuration

The communication configuration is the first tab that the user has to interact with (Fig. 17). This tab provides users with four fundamental features: initiate a communication, close a communication, discover the TIMs available in our sensor network and discover the transducerChannels available in a selected TIM.

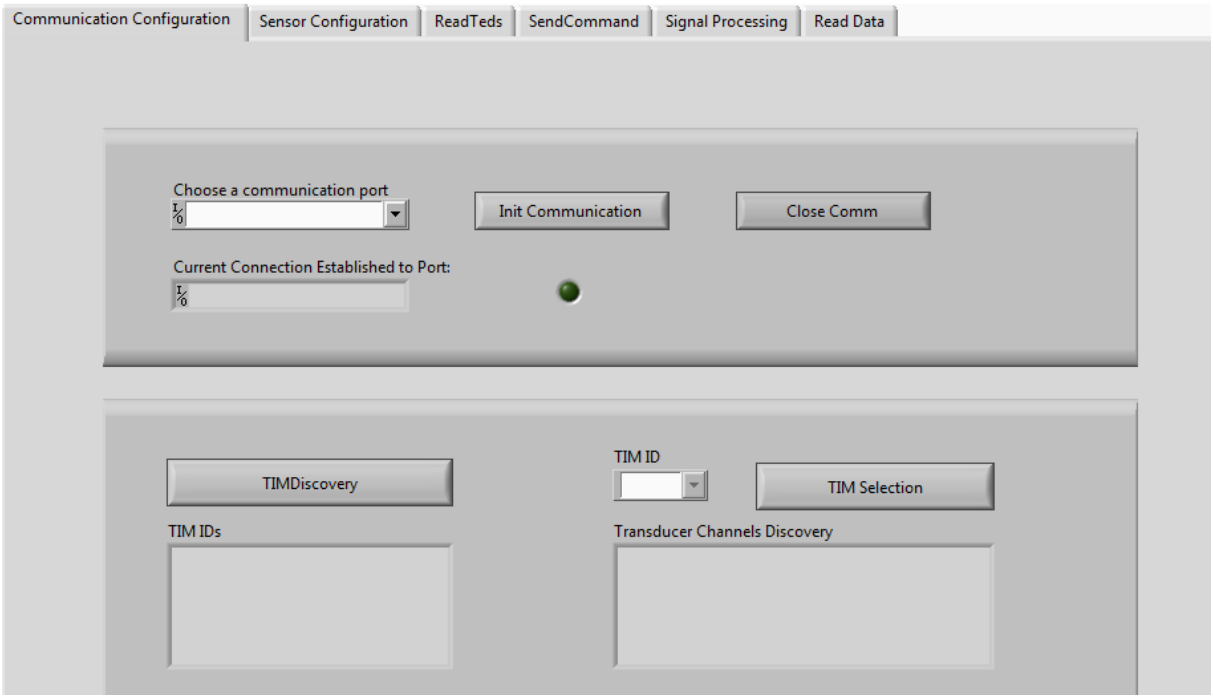

Fig. 17 Communication Configuration Tab

3.2.2.1.1 Initiate a Communication

Through the *Choose Communication Port* selector the different COM ports available in our machine are shown. In order to initiate the communication with an NCAP, the user has to choose the COM port to which the NCAP has been connected (Fig. 18). Once selected the correspondent COM port, we can initiate the communication by clicking *Init Communication* button.

Fig. 18 Communication Port Selection

As already explained previously, the event of clicking the *Init Communication* button will be handled by our state machine and the code to establish the serial connection will be executed. The serial connection code must be configured with the proper parameters to make it work properly (Baud Rate, Timeout, Parity…). The values used for these parameters can be found in Fig. 19

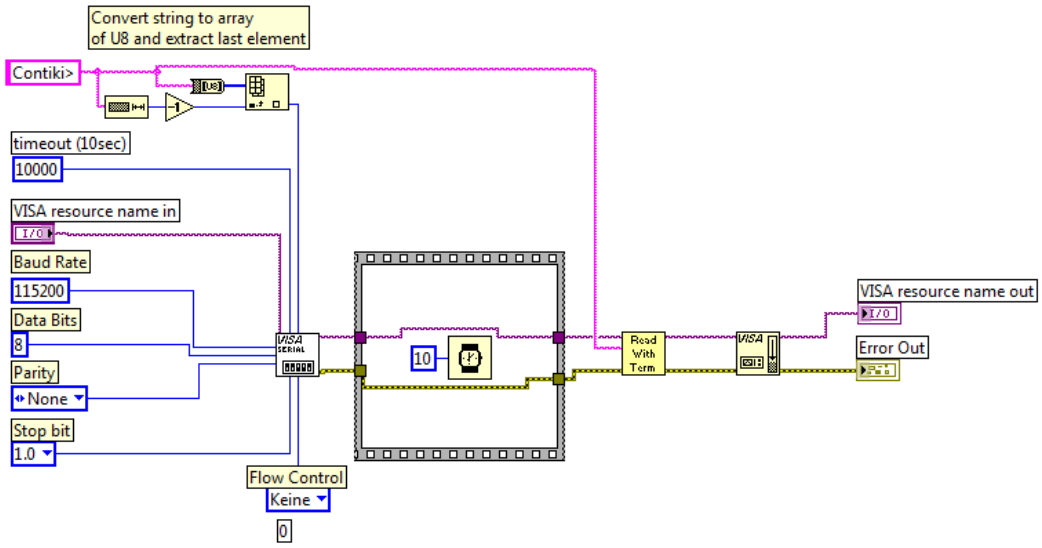

Fig. 19 Serial Communication Configuration

Once established the serial communication using the *VISA serial* VI and after a short delay, the application has to wait to receive a prompt from the NCAP indicating that it is ready to receive commands. To do so, the function *Read With Term* was created to read from the serial port until a termination string is found. In this case, the termination string is the Contiki prompt ("*Contiki>"*). After receiving the prompt, we clear the serial line buffer for any extra undesired characters that can corrupt future serial port reading and everything is ready to send commands to the NCAP. In the front panel, the current connection is shown and the green LED is turned on (Fig. 20).

Fig. 20 Connection Established

3.2.2.1.2 Close Communication

Next to the *Init Communication* button we can find the *Close Comm* button. This button takes the program to the *Close Comm* state. In this state the connection is closed using the *VISA Close VI*. The front panel shows no connection established and the green LED is turned off.

3.2.2.1.3 TIM Discovery

Once a connection is established with an NCAP we can start performing different actions in order to interact with the sensor network. The first step is to find out the TIMs connected to the WSN. To do so the user has to click on the *TIMDiscovery* button in the front panel. This buttons leads the state machine to the *TIMDiscovery* state, where a *TIMDiscovery* command is sent through the Contiki shell through the *VISA Write* VI. The response to this command is an uint16 array of the TIM IDs connected to the WSN, which are presented in a string indicator (Fig. 21)

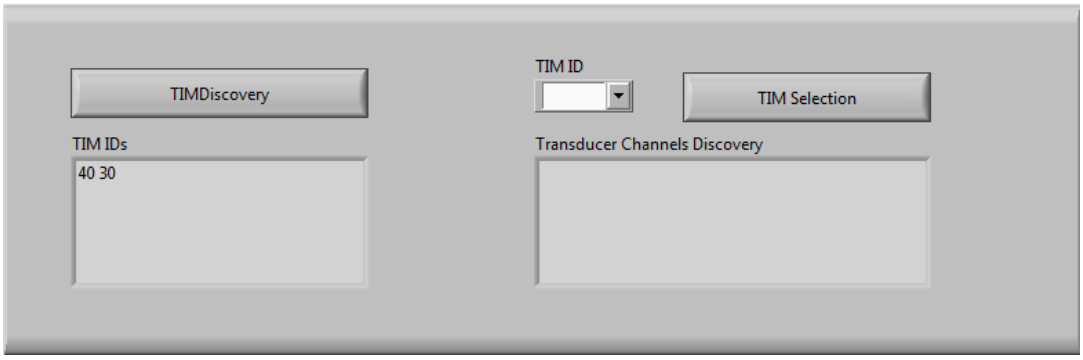

Fig. 21 TIMs Discovered

In the example of Fig. 21 two TIMs (IDs: 32 and 30) have been discovered.

3.2.2.1.4 TIM Selection and TransducerChannel Discovery

Once known the TIMs connected to the WSN, the user can select one of these TIM (Fig. 22) and discover the TransducerChannels available in that sensor node clicking on the *TIM Selection* button. Once the button is clicked, a *TransducerChannel Discovery* command is sent through the Contiki-shell. The response is a uint16 array of TransducerChannels available in that TIM. In the example of Fig. 23 we can see that 6 transducerChannels (0, 1, 2, 3, 4 and 5) have been discovered in the TIM selected (TIM ID: 30)

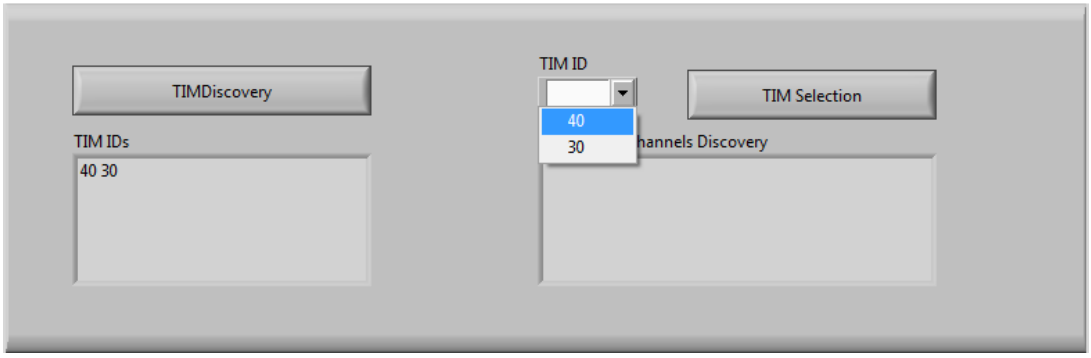

Fig. 22 Select TIM

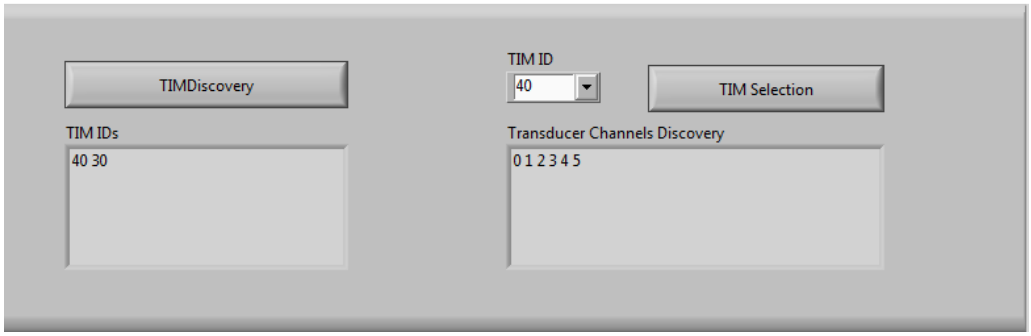

Fig. 23 TransducerChannels Discovery

3.2.2.2 Sensor Configuration

The sensor configuration tab allows users to configure the parameters regarding the sensor data acquisition. These parameters are:

- Number of samples to be acquired
- Sampling period
- ADC channel to be read

This tab makes it really easy to configure the sensor without the need to know exactly what commands of the Contiki-shell should be used.

The number of samples to be acquired is set using a *sendCommand* instruction to the transducerChannel that control the number of samples to be received. The sampling period and the ADC channel are set using a *writeData* command to their corresponding transducerChannels. The data sent using the *writeData* command is previously encoded in base64 format.

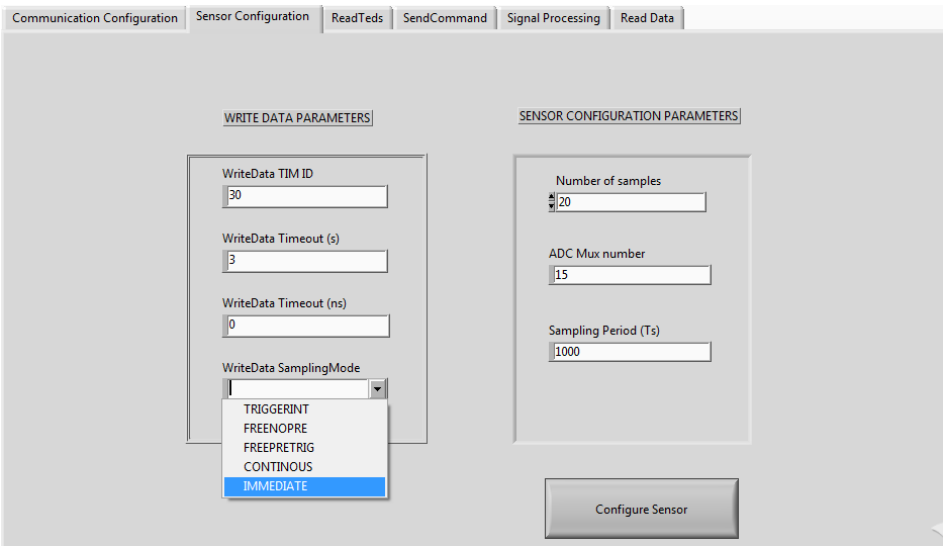

Fig. 24 Sensor Configuration Tab

3.2.2.3 Read TEDS

In the ReadTeds tab (Fig. 25) the user can obtain information of the TEDS of a transducerChannel of a certain TIM. Once filled the necessary parameters to send the command and click the *ReadTEDS* button, a *ReadTEDS* command will be sent to the NCAP. The response of a *readTEDS* command is shown in a string indicator. In Fig. 25, for instance, there can be seen the information of the *MetaTEDS* of the transducerChannel 0 of the TIM 30. The response of the *readTEDS* command is more complex than the ones explained in previous sections. It has the following structure:

Type of Dictionary (uint8)	Number of Argument Arrays (uint8)	Names ArgumentArray	Values ArgumentArray

Table 7 Structure of the readTEDS response

The *type of dictionary* contains a value that indicates how the response data has been structure, i.e. which dictionary has been used. At the moment this variable has no use in the system and the application simply ignores it. The *Number of Argument Arrays* is a uint8 that indicates how many Argument Arrays follow. In the case of a readTEDS response, this value is always 2, since we have an Argument Array for names (fields of the TEDS) and an Argument Array for the values (values of the fields).

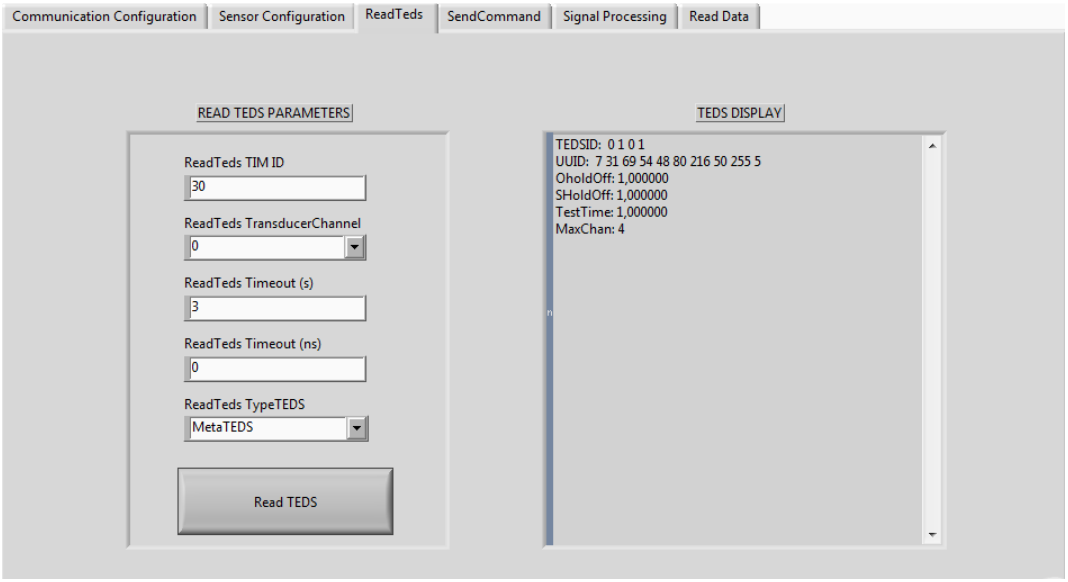

Fig. 25 Read TEDS tab

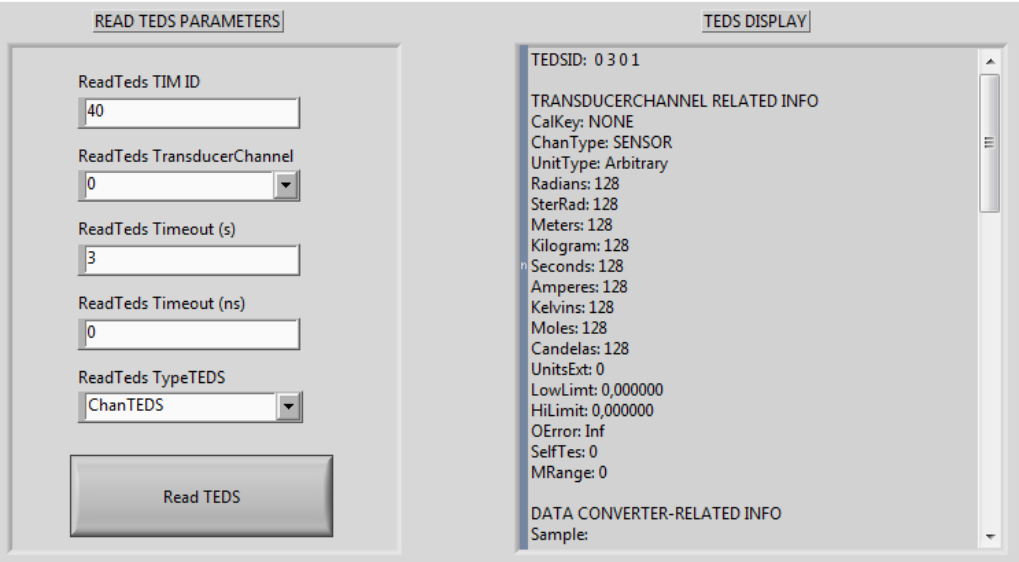

Fig. 26 TransducerChannel TEDS

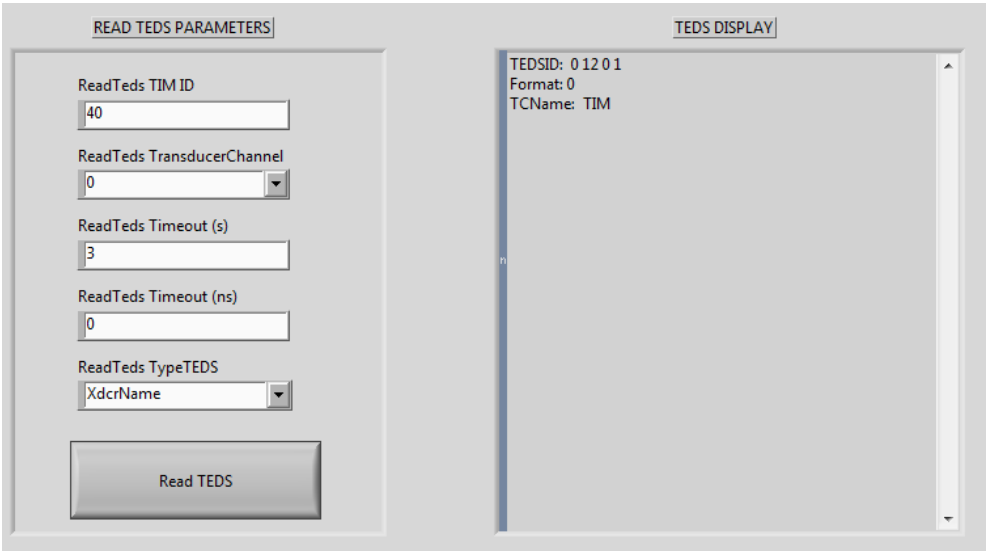

Fig. 27 Transducer Name TEDS

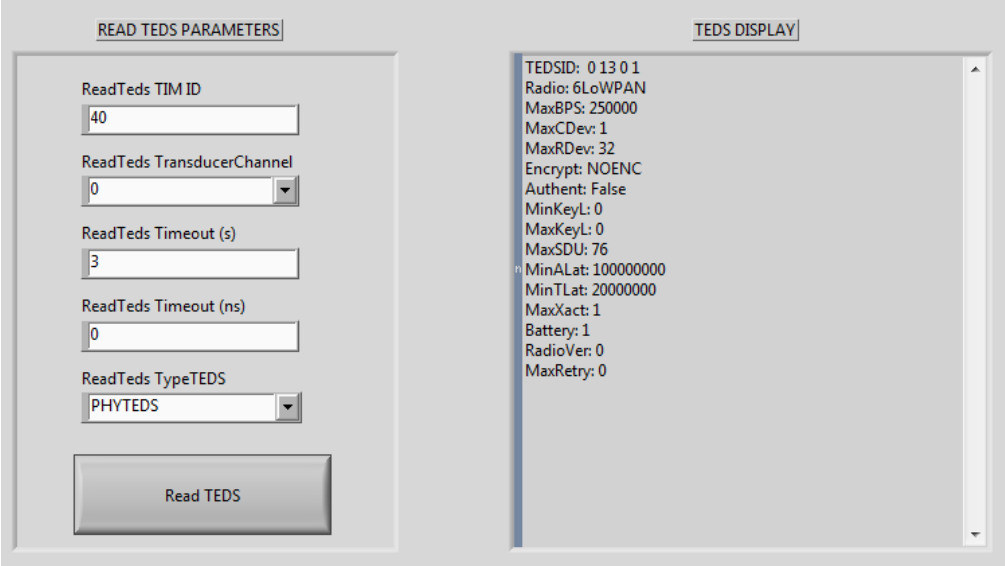

Fig. 28 PHY TEDS

The *Names Argument Array* begins with a uint16 variable that indicates how many names are contained in the Argument Array. After this value, the array contains the different fields (names) of the TEDS in ASCII format and separated by zeroes, so our application can easily parse the different fields knowing the total number of them and parsing the array each time a zero is found.

Number of Fields (uint16)	Field 1	0	Field 2	0	…	0	Field N

Table 8 Structure of the Names Argument Array

After the *Names Argument Array*, we receive the *Value Argument Array*, which contain the values of the previous fields. Each value contained in the array is structured in a TLV format:

Type	Length	Value

Table 9 Structure of the elements of the Value Argument Array

The *type* field specifies what kind of value follows (uint8, uint16, octet array...). If the *type* is an array, the *type* field is followed by another field that contains the length of the array. Finally, the *value* field contains the actual value in the specified format.

The position of each name corresponds to the position of its value, so the first value corresponds to the first name, the second value to the second name, and so on. Therefore, once our application decodes the names and the values in two different arrays, it is trivial to join them in a single array where each field and value is presented in string format using the tools available in labVIEW.

3.2.2.4 Send Command

In the SendCommand tab of the GUI the user can send commands to a certain transducerChannel of an specified TIM once the *Send Command* button is pressed. The commands specified in the IEEE1451.0 standard can be found in section 7 of the document [9]. However, the code used in our sensor network implements only certain commands. More specifically, the code available only accepts commands that have as argument a uint16 value. Furthermore, the code does not provide with any response to the user. Therefore, the S*end Command Feedback* that can be seen in in Fig. 29 will only show the user the structure of the command sent to the NCAP.

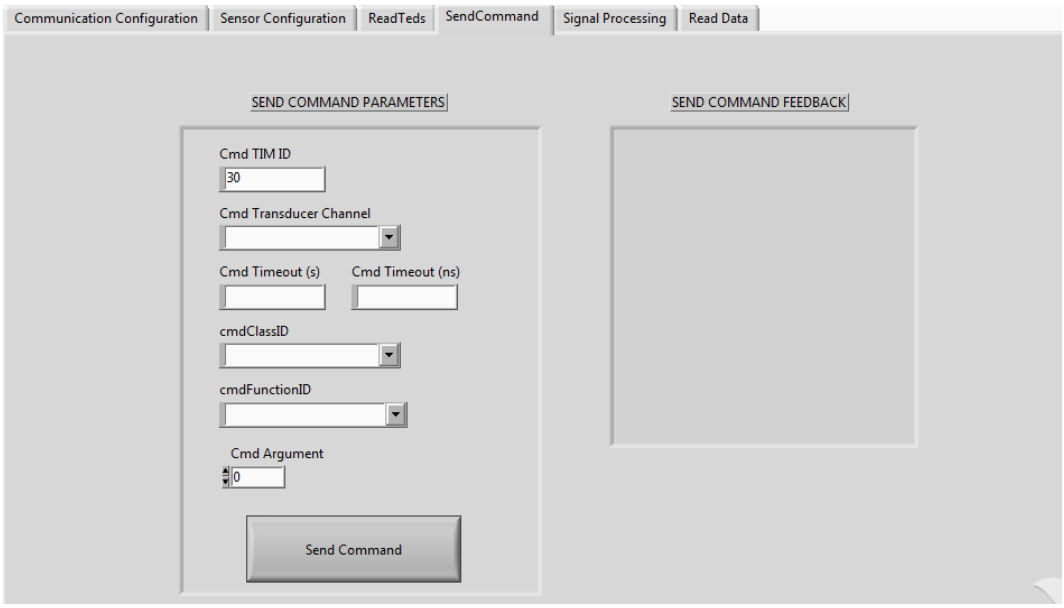

Fig. 29 Send Command Tab

The command functions ID that can be chosen by the user depend on the command Class ID previously selected. For this reason, depending on the *cmdClassID* introduced, the user will have different *cmdFunctionID* options. This is done thanks to the *event structure*, which is listening to the events in the *cmdClassID* selector. When there is a value change, the item list of the *cmdFunctionID* is also changed accordingly.

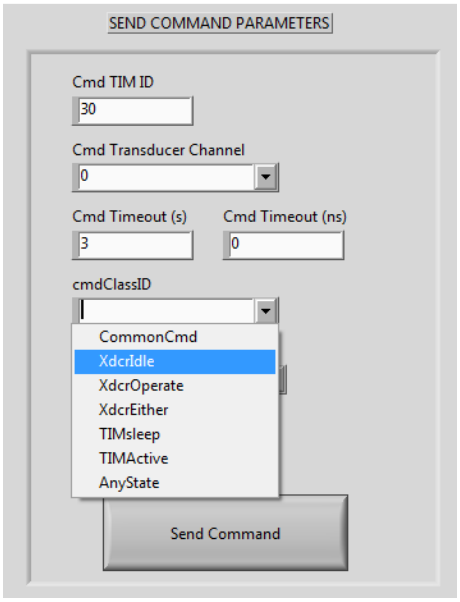

Fig. 30 Choose a Command Class ID

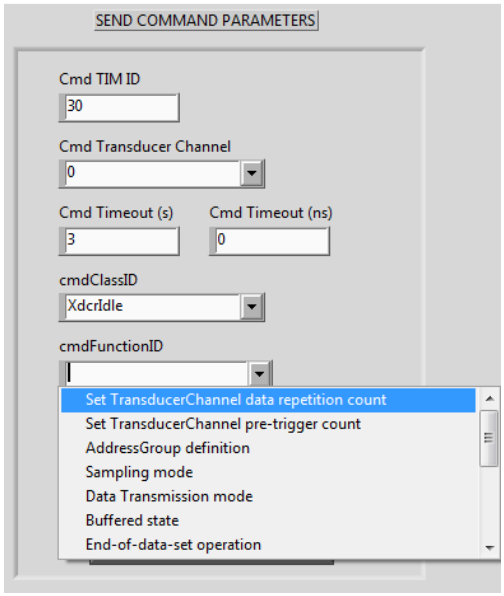

Fig. 31 Choose Command Function ID

3.2.2.5 Signal Processing

The Signal Processing tab of the front panel provides users with the possibility to design in an interactive way, different kinds of fixed point digital FIR filters. These filters can be designed in an interactive way thanks to the tools provided by labVIEW. The types of filters that the user can implement are described in section 2.5.1.2. Once the user designs the filter introducing the desired parameters, the designed filter and its fixed point version can be analysed through its magnitude, pole-zero diagram, impulse response, phase response and coefficient report where both the floating point and the fixed point versions of the filter are compared. The coefficient report allows for checking the quantized values of the coefficients.

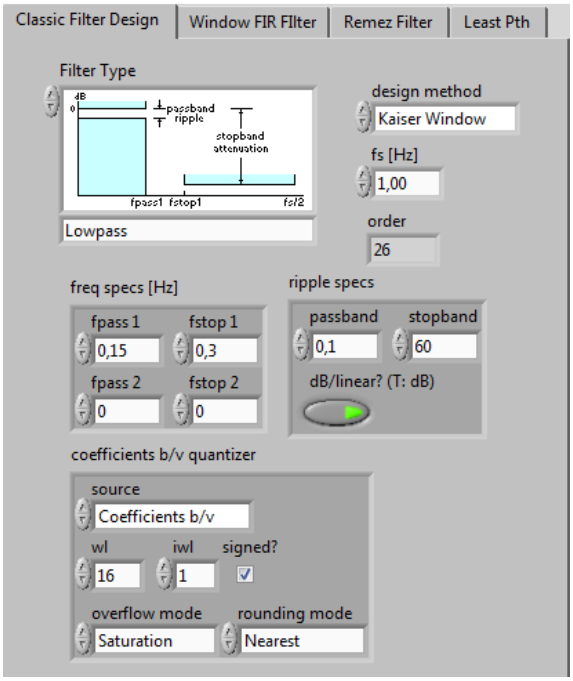

Fig. 32 Filter Design Specifications

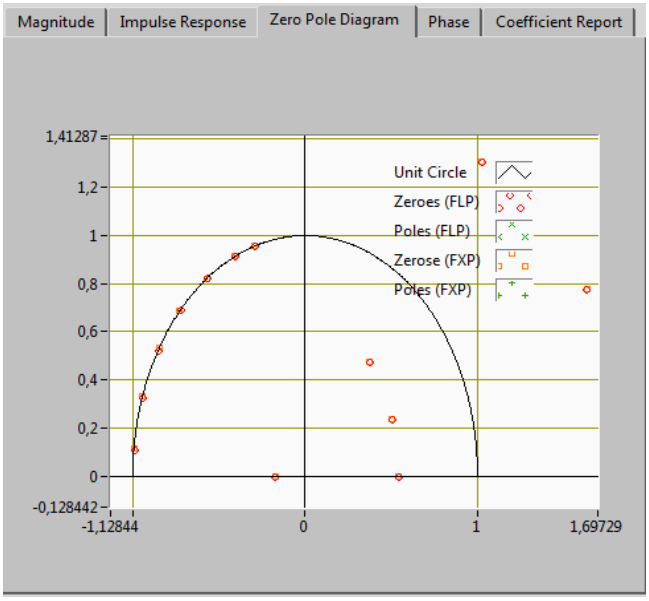

Fig. 33 Pole-Zero Diagram

Once the analysed filter is considered correct, the user can apply the filter to a certain TIM clicking the *Apply Filter* button present in the tab. Signals coming from that TIM will be filtered with the applied filter. The user must then *readData* from that TIM to check if the behaviour of that filter is the expected. Note that due to fact that we are using fixed point filters; sometimes the behaviour of these filters can change, for example, due to the overflow that may appear during the filtering computation.

In case the user wants to deactivate the filtering in a certain TIM, it is necessary to configure the sensor again, which returns the option of filtering to the default value, which is no filtering activated.

3.2.2.6 Read Data

In the Read Data tab, the user is capable of reading data from a sensor. After specifying the desired source of data (TIM ID and TransducerChannel), the timeout and the sampling mode, the user can collect the data pressing the *Read Data* button. Note that the data received will depend on the sensor configuration defined in the sensor configuration tab (3.2.2.2).

This tab shows a graph of the data received and other relevant parameters of it such as the arithmetic average, the standard deviation or its maximum value. The user has also the possibility to study the frequency spectrum of the received signal, which is calculated through an average FFT function provided by LabVIEW. Finally, users can also analyse the statistical behaviour of the received signal through its histogram. It is important to note that users will be able to export all data received to another platform, such as excel, for data storage or further analysis.

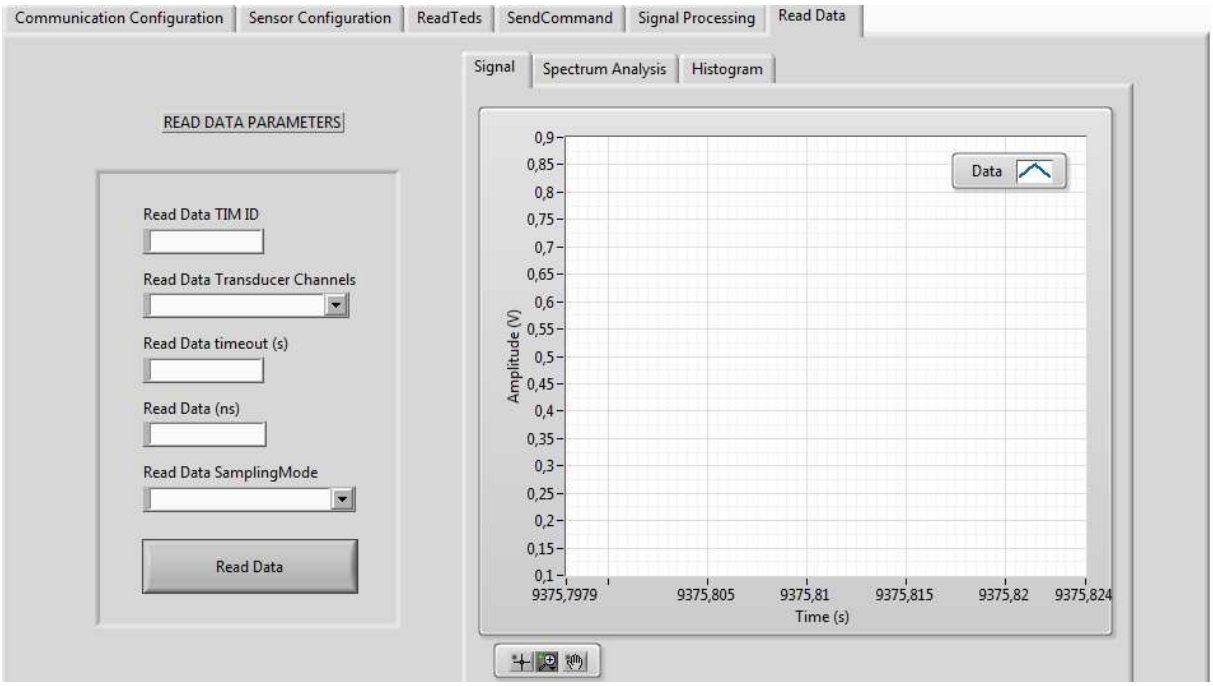

Fig. 34 Read Data Tab

The response of the *Read Data* command sent through the Contiki-shell is structured in blocks of 10 bytes (5 words). Each block is structured as follows:

Timestamp *sec* (uint32)	Timestamp *ns* (uint32)	*Value* (uint16)

Table 10 Structure of a data block

With this structure, our program can easily parse the data received and obtain two different arrays, one for the time and one for the value, which will allow for a representation of the signal in the front panel. The calculation of the time is done using the following procedure:

$$Time = sec + ns \cdot 2^{-9} \quad (6)$$

In Fig. 35 we can see how our GUI displays a sinusoid received from a certain TIM. In Fig. 36 and Fig. 37 we can see its frequency response and its histogram, respectively

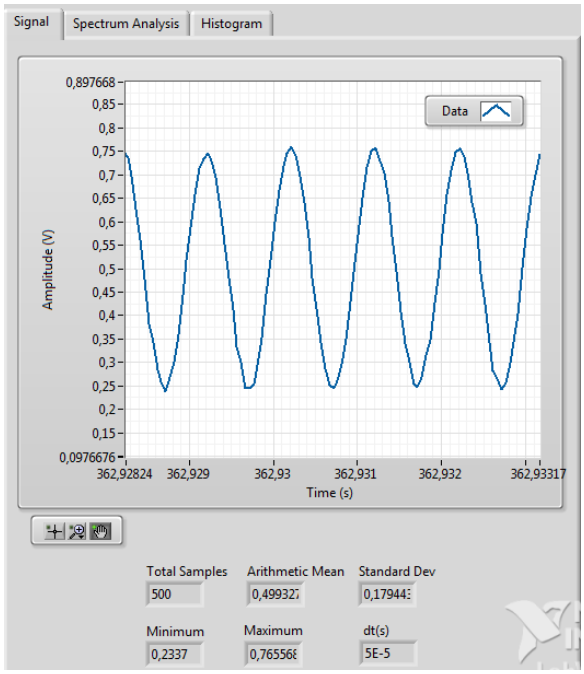

Fig. 35 Sinusoide Signal

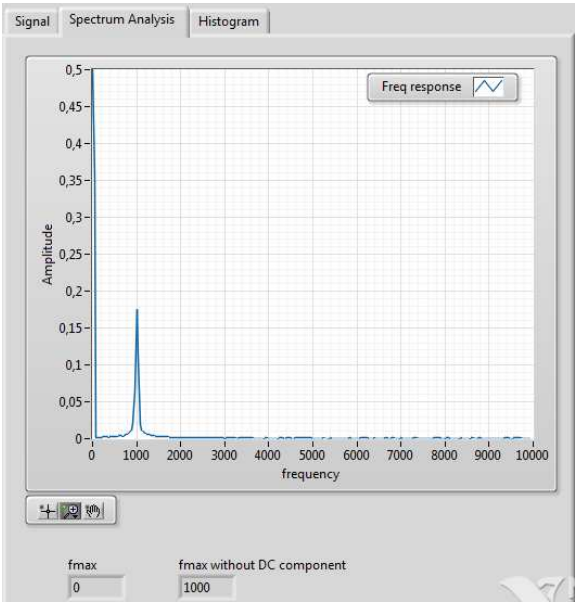

Fig. 36 Sinusoid Frequency Response

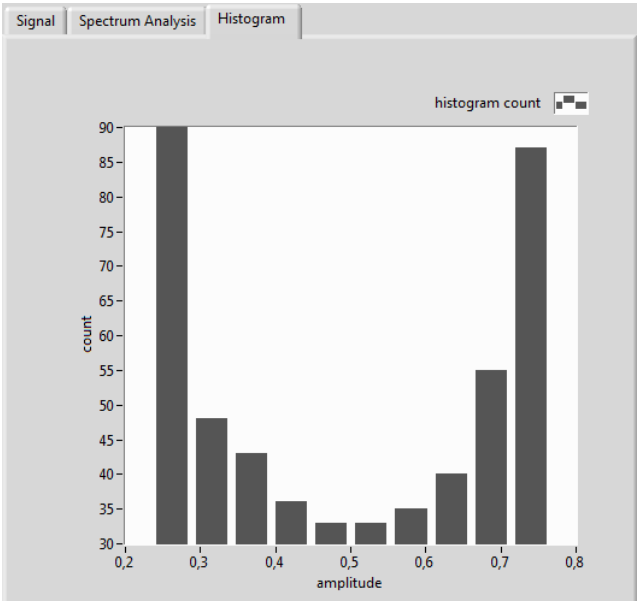

Fig. 37 Sinusoid Histogram

3.3 Error Handling

Error handling is a delicate part of any complex program. In our user interface, we can differentiate two sources of errors: errors produced by the sensor network and errors occurred in the computer side. It is to be noted that there are other errors that are automatically handled by LabVIEW.

Thereferoe, we will not be discussing about them. These errors are, for example, introducing invalid parameters when designing a filter.

3.3.1 Errors received in command responses from the NCAP

The responses of the commands (*ReadTEDS, ReadData*…) received by our application begin with two bytes (16 bits) that correspond to the error code. This error code can be produced by many different reasons (TEDS not implemented, TIM ID not available etc.). When an error occurs, our user interface decodes this error and shows a dialog box indicating what kind of error has occurred. The structure of the error code implemented in our system is shown below:

Bits	Use
15 through 13	Error Source
12 through 0	Error Code Enumeration

Table 11 Error Code Structure

Value	Source of Error
0	Unknown source of error
1	Error from the local IEEE 1451.0 layer
2	Error from the local IEEE 1451.5 layer
3	Error from the remote IEEE 1451.5 layer
4	Error from the remote IEEE 1451.0 layer

Table 12 Error source possible values

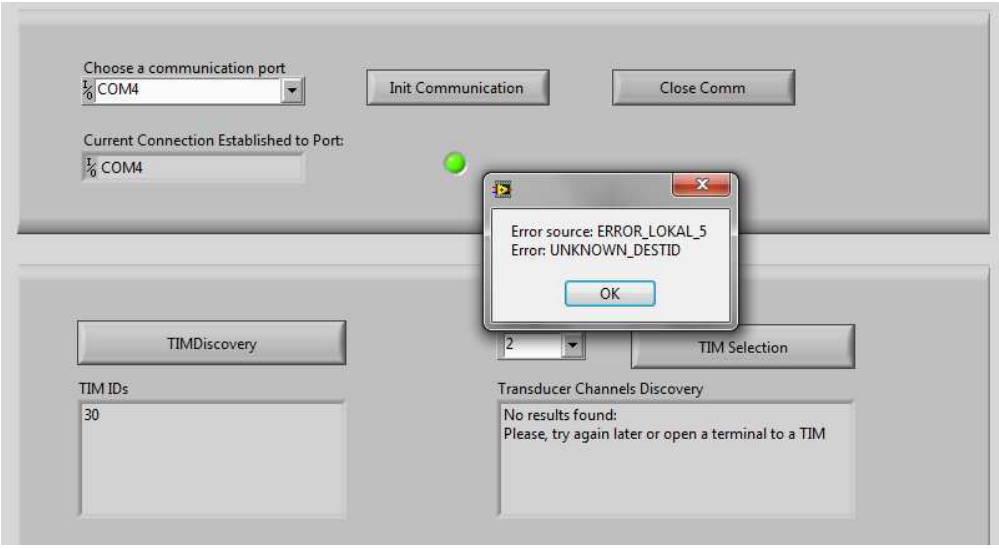

Fig. 38 Error Dialog Box

3.3.2 Errors originated in the GUI code

To the extent possible, the code of the GUI tries to avoid errors controlling the actions carried out by the users. An example is presented in Fig. 39, where the application warns the user when trying to discover TIMs without any connection established to an NCAP

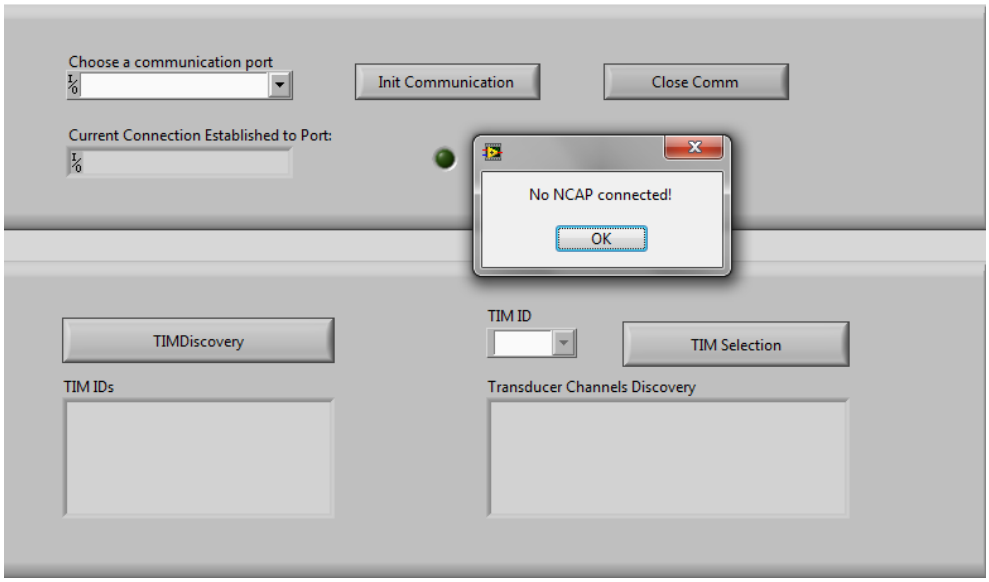

Fig. 39 Dialog Box with improper action warning

However, there are errors that cannot be avoided and have to be handled correctly. In our case these errors are related to the VISA communication functions. These functions may generate errors and these errors must be handled. The application is designed in a way that if an error is detected after a VISA function action, the next state is automatically *Close Communication*. Therefore, if an error occurs, we rapidly close any communication established and show a dialog box to the user with the error message. The serial communication is closed and the user has to restart the process of establishing again a communication with a NCAP. On the other hand, if there is no error the next state will be *IDLE* and the program continues normally.

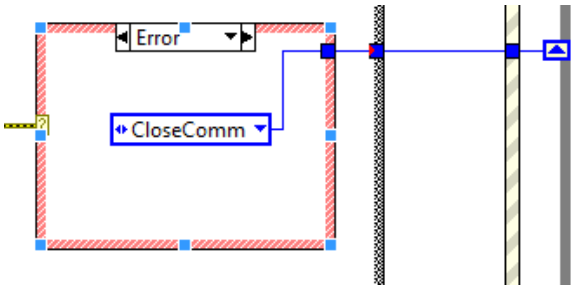

Fig. 40 Communication Error

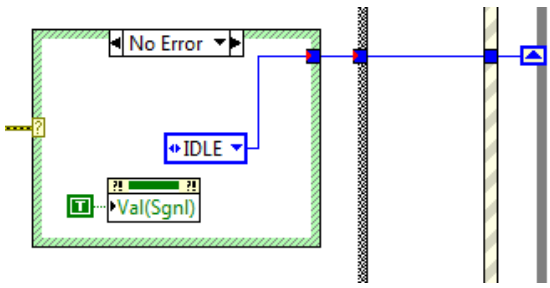

Fig. 41 No Communication Error

3.4 Data Acquisition and Filtering Tests

In order to test and analyse the behaviour of the data acquisition and filtering features, we realized different tests. In this document we will present three significant tests: checking the performance of a filter, low pass filtering and high-pass filtering.

3.4.1 Filter Performance Test

The first test we present in this document consists in checking the performance of a filter designed with the GUI. First of all, we are going to design a filter and analyse its performance with the tools that the GUI provides. In Fig. 42 we can see the parameters introduced for this filter and its magnitude response. With the parameters we entered and the magnitude response seen in the graphic (Fig. 42), we expect our filter to let the frequencies up to 500Hz pass, then start filtering the higher frequencies and eliminate almost completely the frequencies higher than 2 KHz. In the graphic, we can also see that both the floating point and the fixed point versions of the filter have almost the same behaviour (black and red lines) so we can conclude there is no big loss when quantizing the coefficients. However, in the process of filtering in the TIM it is possible to have a different performance due to the problems related to fixed point filtering presented in section 2.5.2.

This test shows the procedure to be followed in order to check the performance of a filter in the frequency domain. The exact test to be carried out by a user might change depending on the final application of the filter.

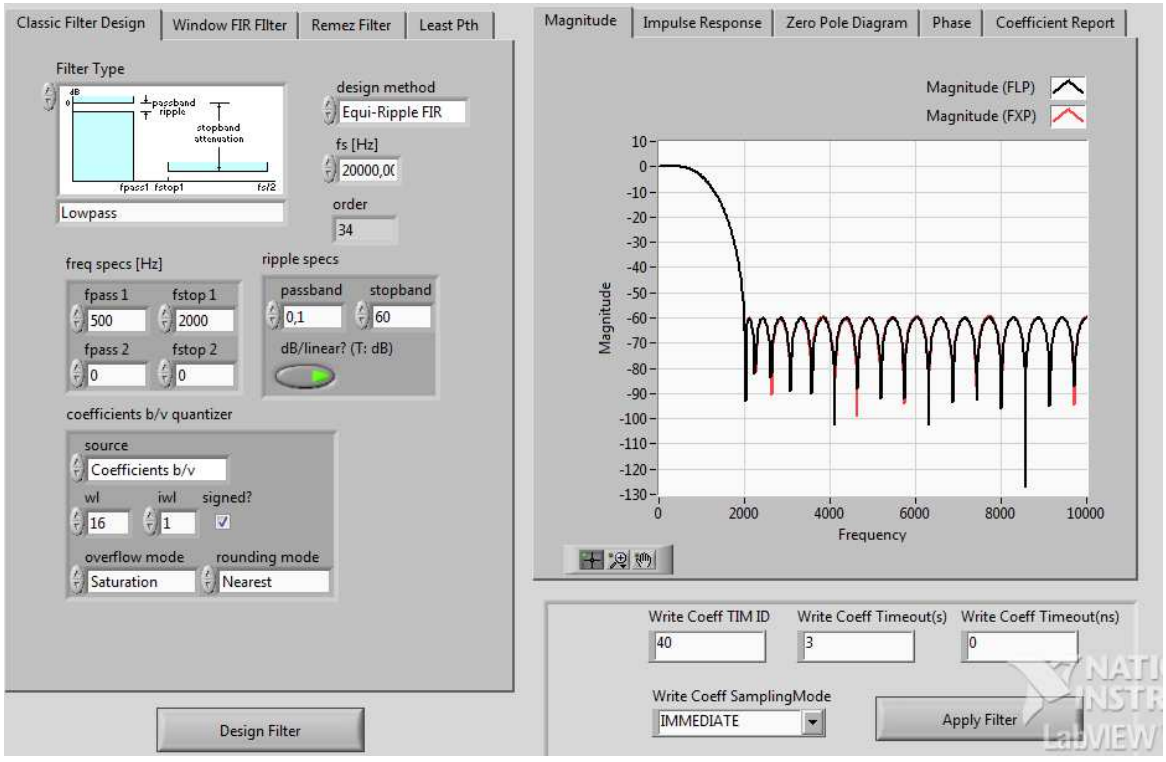

Fig. 42 Test Filter Design

After applying the filter to the TIM, we will acquire sinusoid signals of different frequencies to see the action that the filter carries out on them and if the results are as expected. The acquisition of these sinusoids is done with a sampling period of 50 s and, thus, a sampling frequency of 20 KHz. See below some of the sinusoids applied to the filter and its output once filtered.

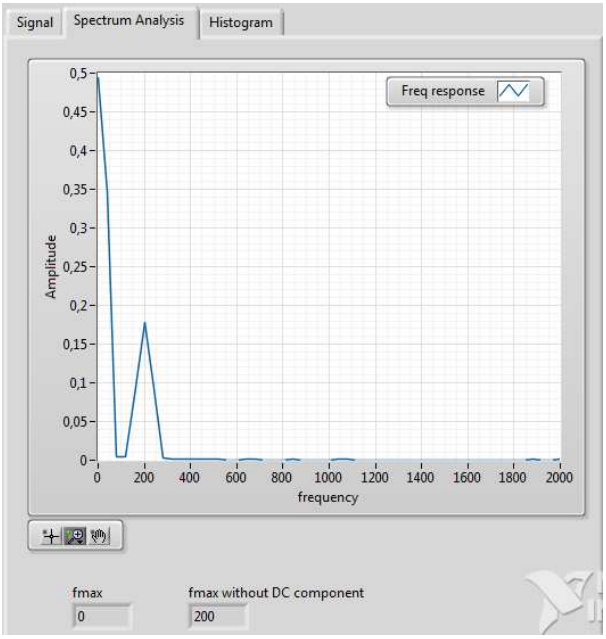

Fig. 43 Sinusoid 200Hz (No Filter)

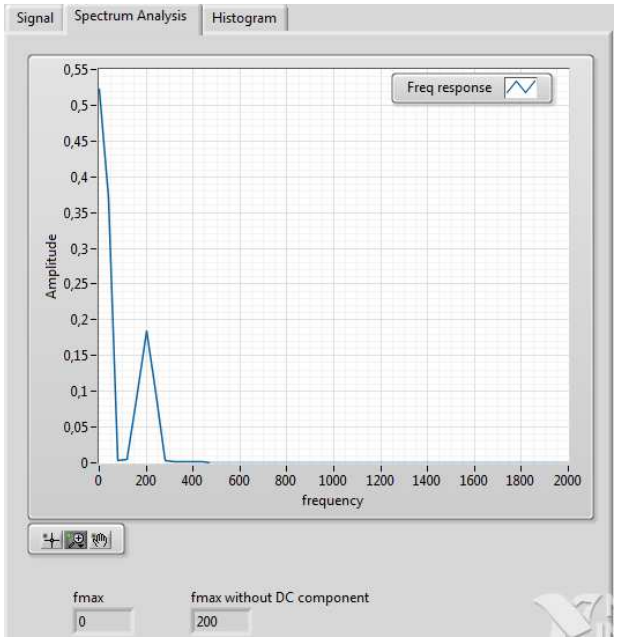

Fig. 44 Sinusoid 200Hz (Filtered)

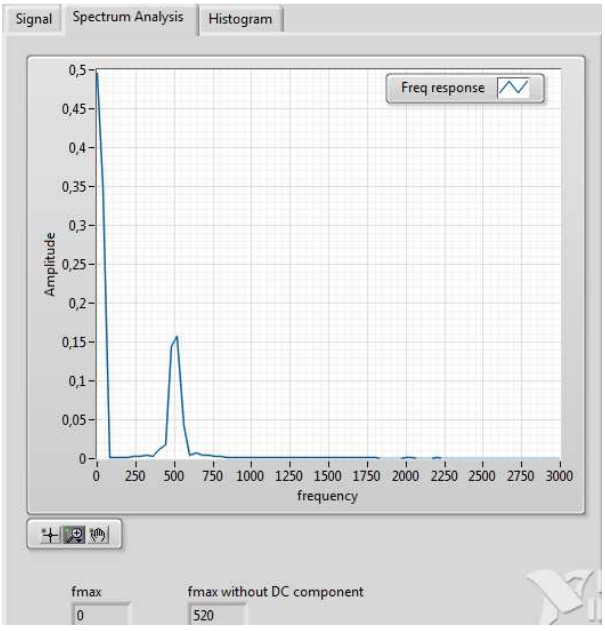

Fig. 45 Sinusoid 500 Hz (No Filter)

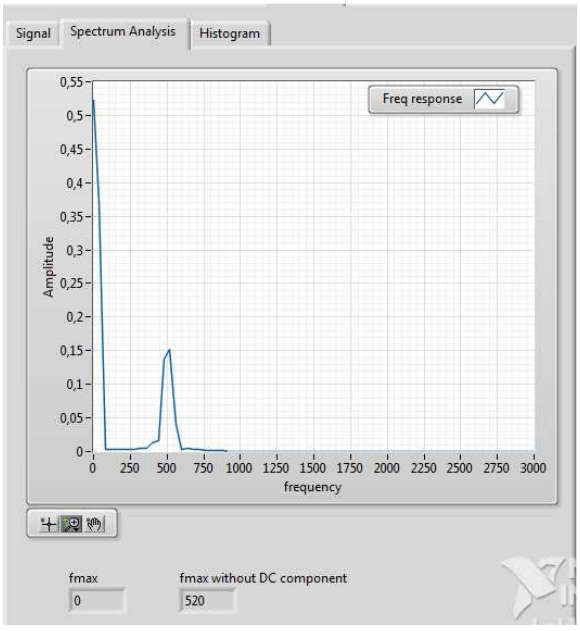

Fig. 46 Sinusoid 500Hz (Filtered)

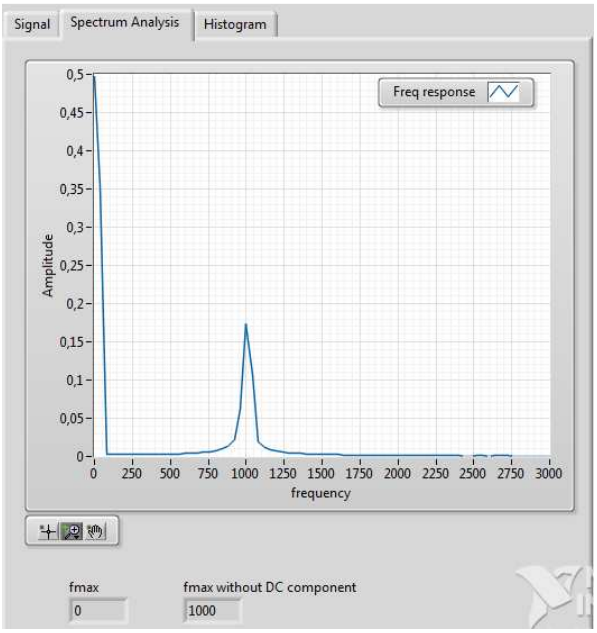

Fig. 47 Sinusoid 1KHz (No Filter)

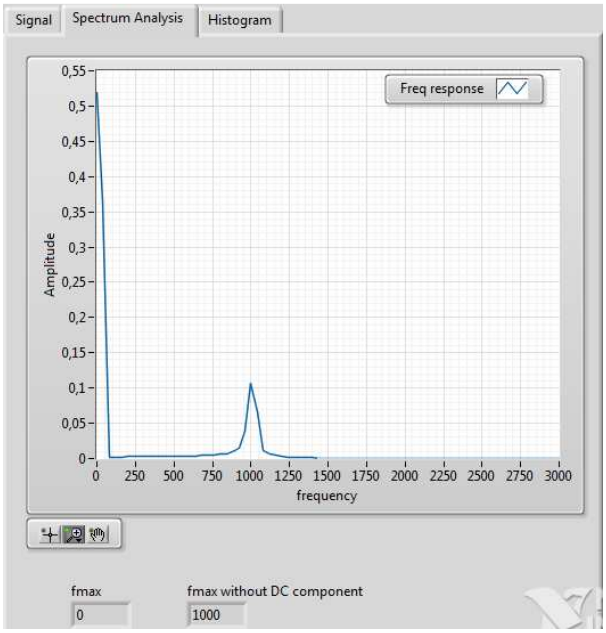

Fig. 48 Sinusoid 1 KHz (Filtered)

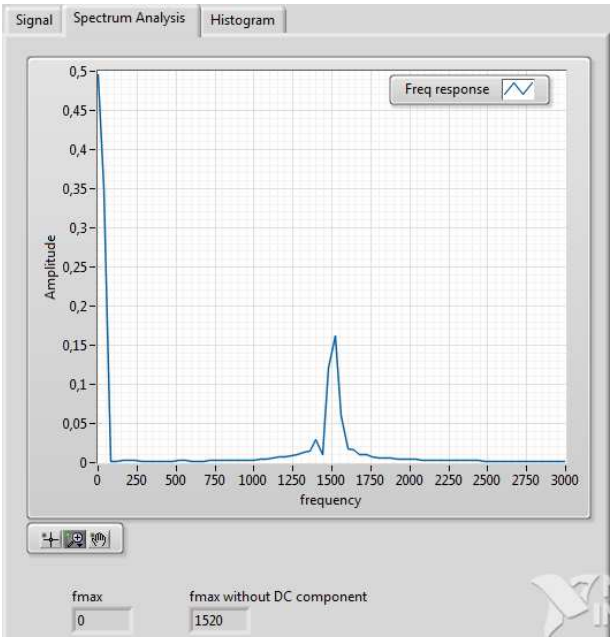

Fig. 49 Sinusoid 1.5 KHz (No Filter)

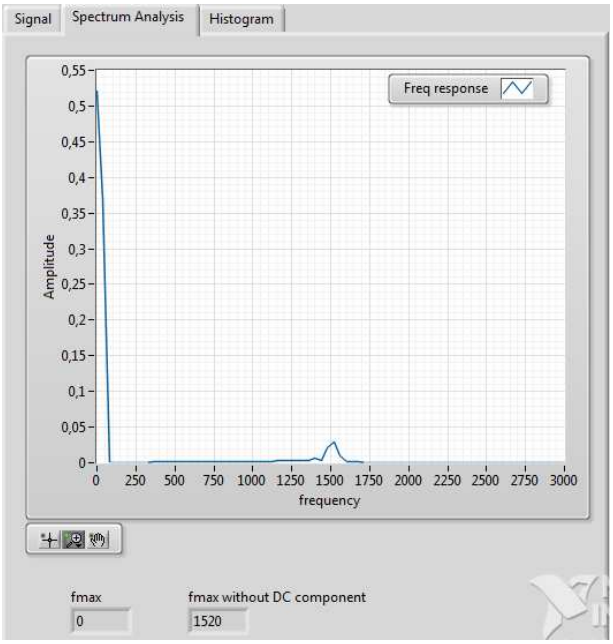

Fig. 50 Sinusoid 1.5 KHz (Filtered)

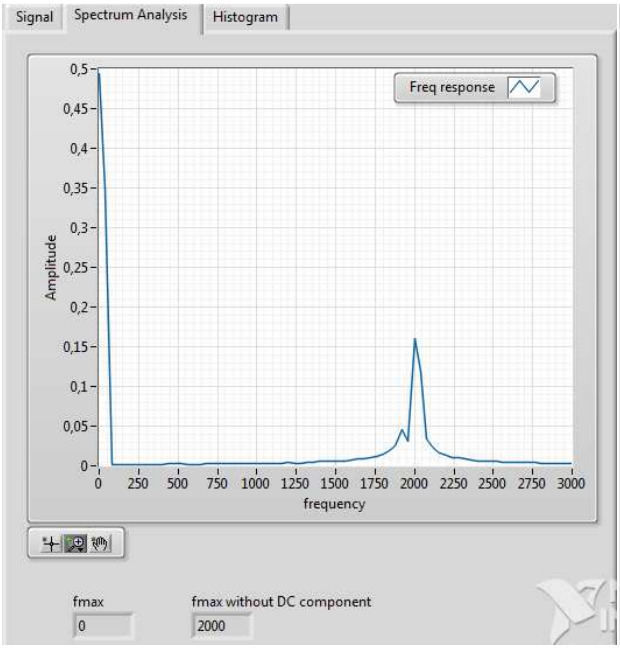

Fig. 51 Sinusoid 2 KHz (No Filter)

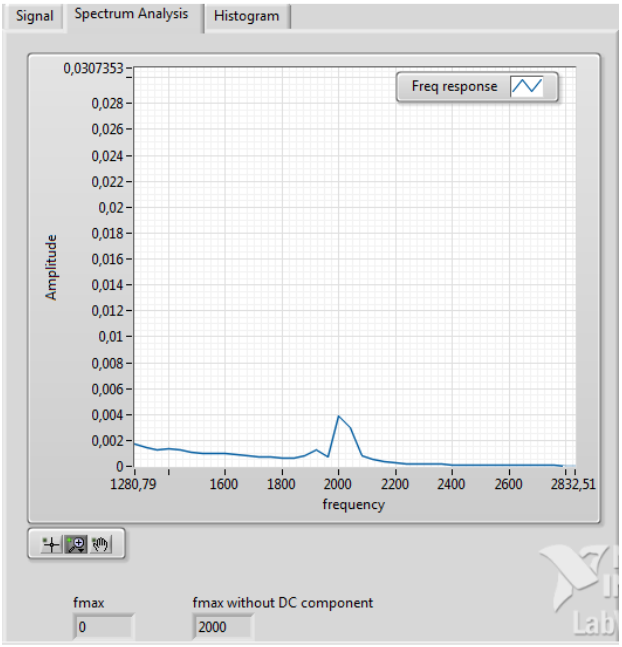

Fig. 52 Sinusoid 2KHz (Filtered)

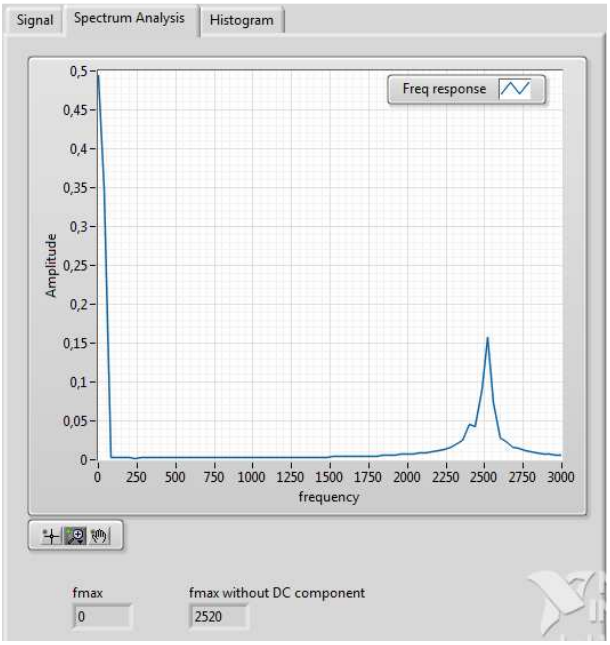

Fig. 53 Sinusoid 2.5KHz (No Filter)

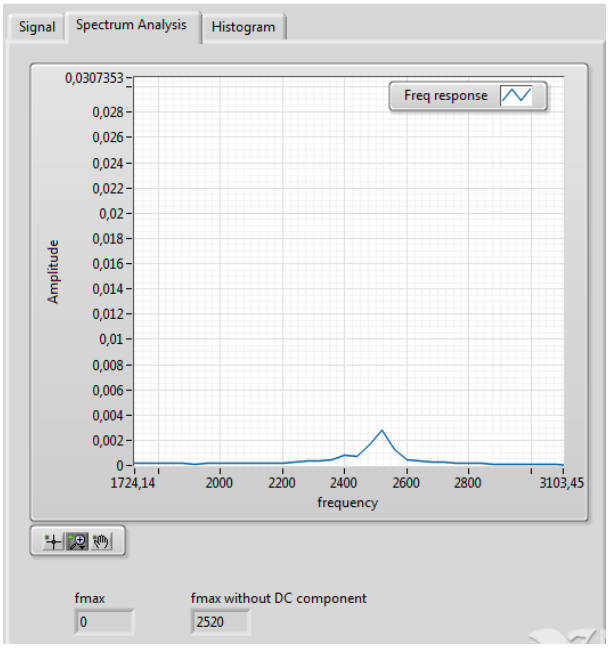

Fig. 54 Sinusoid 2.5 KHz (Filtered)

Analysing the different figures above we can conclude that the behaviour of the designed filter is similar to the one designed in the GUI. Until 500Hz, the signal is not attenuated by the filter. When we reach 500 Hz, the signal begins to be attenuated. In fact in Fig. 46 we can see that the sinusoid at 500Hz suffers already a slight attenuation. When we reach 2 KHz, our stop frequency, the signal is clearly attenuated and sinusoids with higher frequencies suffer also a high attenuation. Therefore, this

filter seems to comply with our specifications. However, for certain applications where high precision is required, a deeper analysis should be carried out taking into account the attenuation levels in the different frequencies and changing the filter specifications until reaching the required results (try and error procedure).

3.4.2 Low Pass Filter Test

In this test we wanted to try to cancel the DC offset of a sinusoid. To carry out this test we acquired a sinusoid coming from a function generator connected to a TIM of our WSN with the following parameters:

- Freq = 1KHz
- Amplitude (Vpp) = 0.5V
- DC Offset = 0.5V
- Ts = 100µs
- Fs = 10KHz
- # Samples = 128

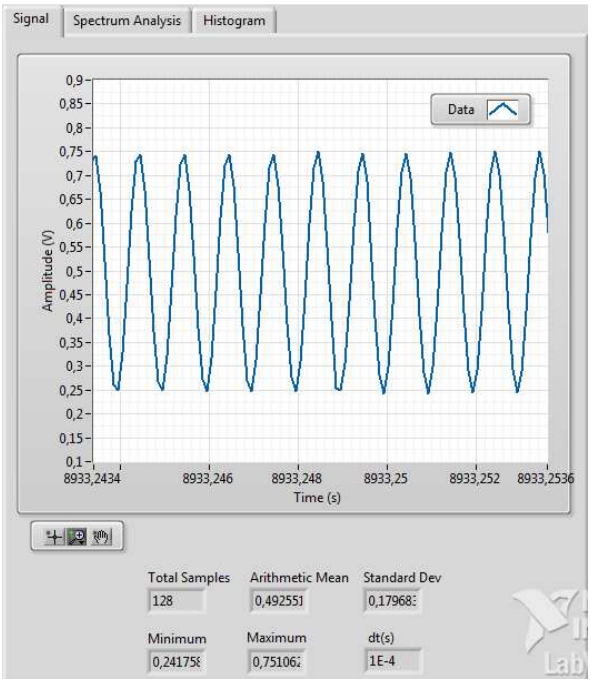

Fig. 55 Sinusoid to filter

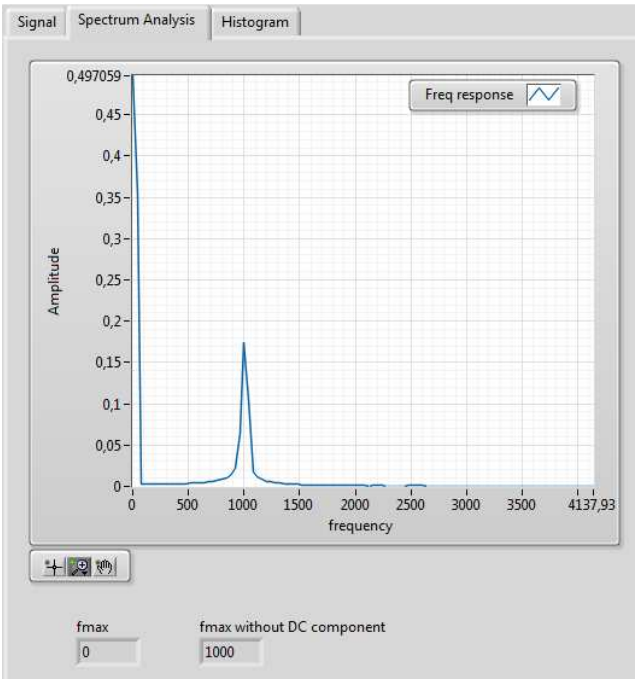

Fig. 56 Frequency Response

We can observe in Fig. 56 that the frequency response is composed of a large DC component and a tone at 1KHz, the frequency of our sinusoid. To cancel the DC offset we need to design a high pass filter. Refer to Fig. 57 for the parameters introduced in the filter design tool of our GUI to design it.

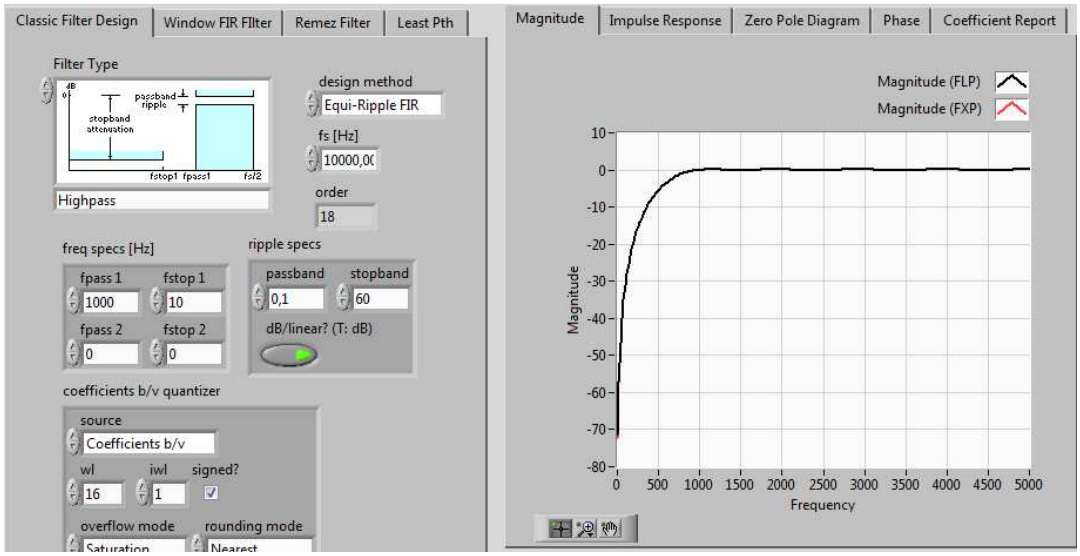

Fig. 57 High Pass Filter Design

We can see that the designed Equi-Ripple FIR filter has a high-pass frequency response type and its maximum is reached at 1 KHz, the frequency that we want to maintain. Once observed that the characteristics of the filter are as desired, it is time to test it with the real signal. See below the filtered signal both in time and in frequency domain.

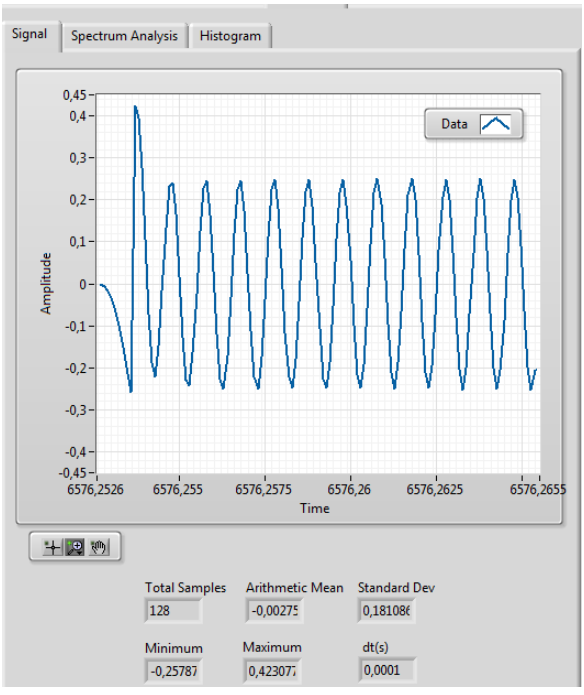

Fig. 58 Filtered Signal(Time Domain)

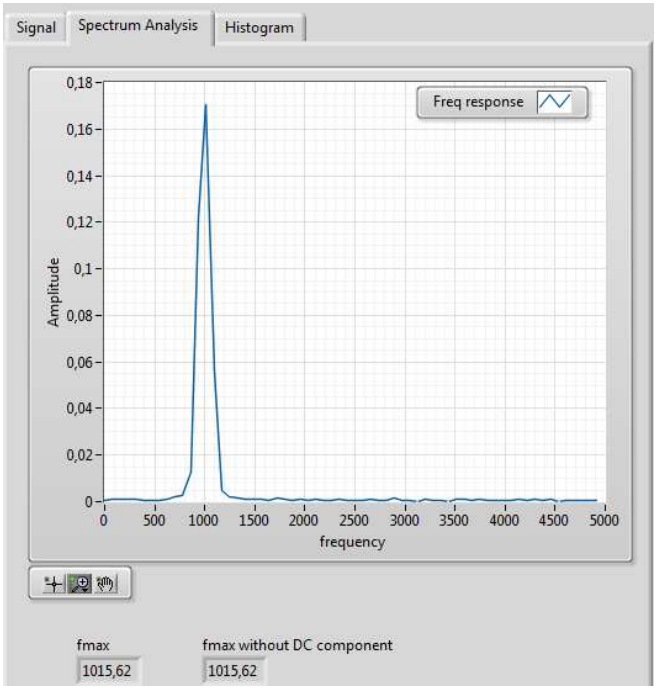

Fig. 59 Filtered Signal (Frequency Domain)

As seen in the figures above, the signal obtained is a sinusoid from -0.25 to 0.25V (although having a strange behaviour at the beginning of the signal). Therefore, we obtained the same sinusoid as before

but without the DC offset, i.e. centred at 0 V. In the frequency domain we can see that now we only have one component at 1 KHz, the single tone that corresponds to our filtered sinusoid.

3.4.3 High Pass Filter Test

Another test was realized to try to filter high frequencies. In this case we used a square signal also generated with a function generator. This square signal had the following parameters:

- Freq = 500Hz
- Amplitude (Vpp): 0.5V
- Offset: 0.5V
- Ts: 100µs
- Fs = 10KHz
- # Samples = 500

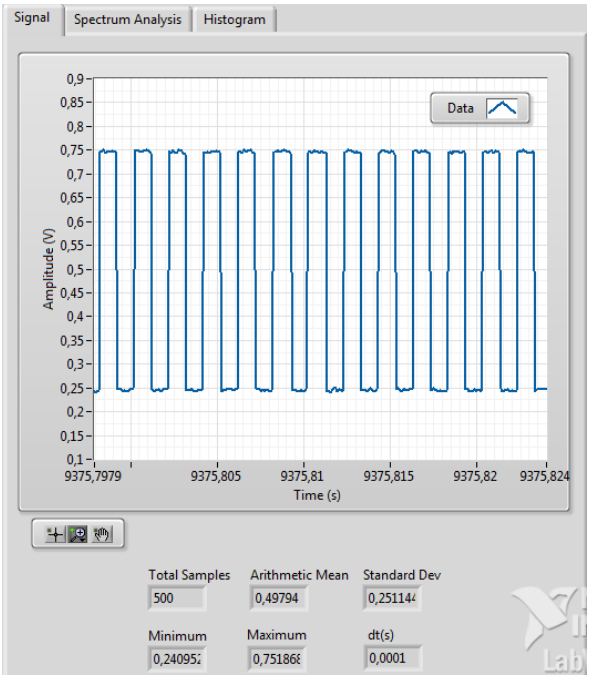

Fig. 60 Square Signal (Time Domain)

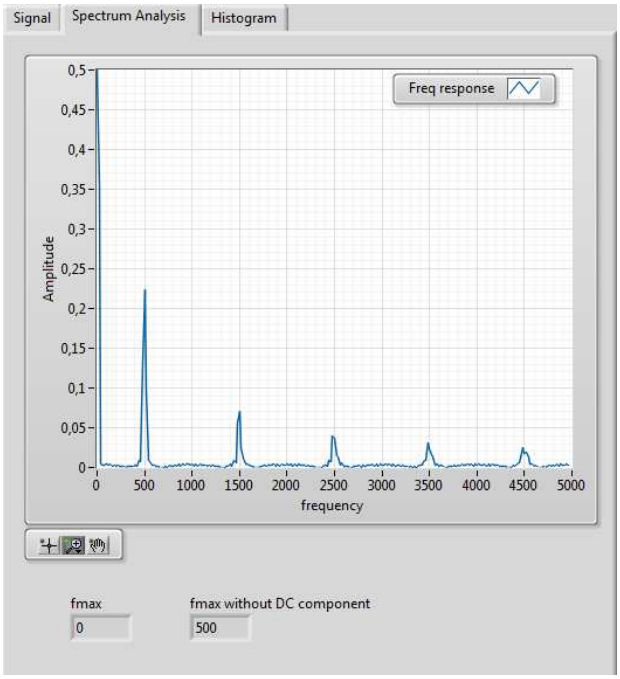

Fig. 61 Square Signal (Frequency Domain)

In the previous figures we can observe the square signal between 0.25 and 0.75V and the offset of 0.5V. In the frequency domain we can see a DC component corresponding to the Offset, a main tone at 500Hz and the harmonics at 1.5 KHz, 2.5 KHz etc.

We want to design a filter that eliminates the high frequency harmonics in order to obtain only a tone at 500 Hz. To do so, the following filter was designed:

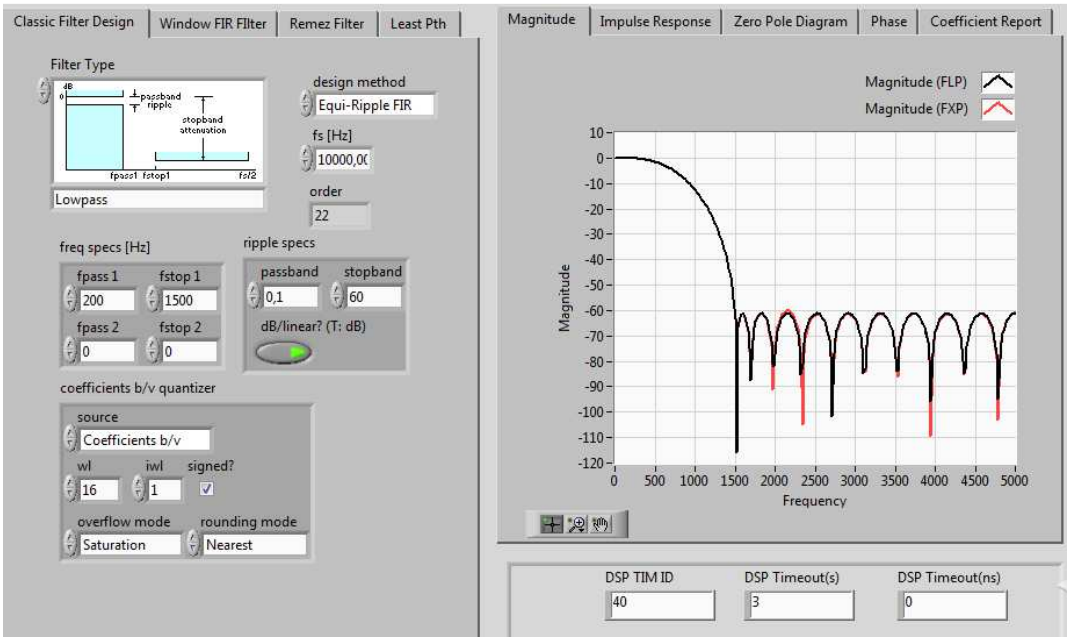

Fig. 62 Low Pass Filter Design

We can see that the filtered design eliminates high frequencies and allows our low frequencies to pass. After applying this filter to the square signal, we obtain a single tone at 500 Hz and the DC component, obtaining therefore a sinusoid of 500 Hz (Fig. 63), with an offset of 0.5V and with amplitude of 0.5 V (Vpp). We can observe in the frequency response how the high frequencies are completely eliminated (Fig. 64).

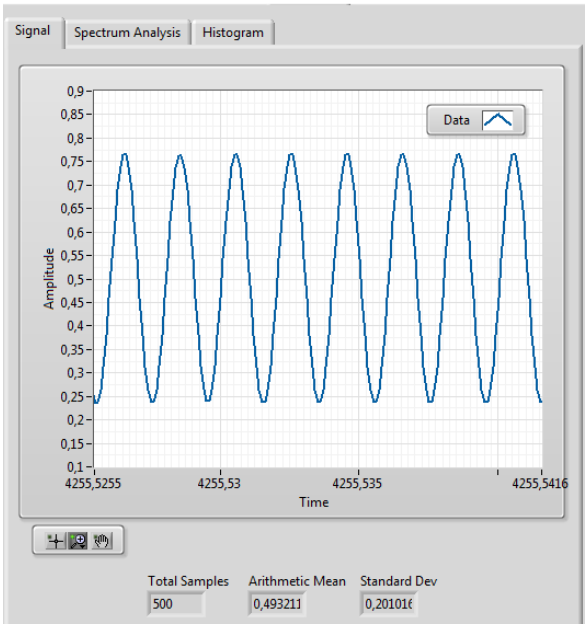

Fig. 63 Square input filtered (Time Domain)

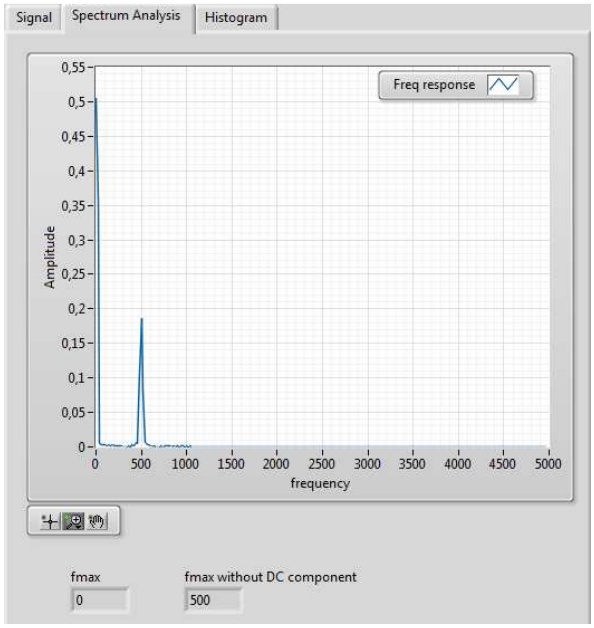

Fig. 64 Square signal filtered (Frequency Domain)

4 Conclusions

After working in this project and observing the results accomplished in the previous sections, we can conclude that it has been possible to create a user-friendly interface that allows users to interact with the IEEE1451.0 layer present in the NCAP without a deep training on the sensor network structure. This project is, therefore, a step towards the complete abstraction of the sensor network in order to allow users to make use of it for different kinds of applications without the need to worry about the network that connects them with the sensor.

In terms of the structure of the GUI, it has been proved that the event structure state machine is a useful method to create well-defined user interfaces with a clear and flexible structure. New features to be implemented in the program only require the creation of new states or the modification of the existing ones.

Regarding the development environment, we can conclude that labVIEW meets the necessary requirements to accomplish the goals set for this project (send commands, acquire and analyse data, design fixed point filters...). Therefore, the work realized in this bachelor thesis can set the basis of a series of GUI releases (versions) that adapt the code to improvements or new functionalities of the WSN in the MDT. Furthermore, new releases can include more functionalities that labVIEW offers to developers, if regarded necessary. In addition, the fact that labVIEW allows for the creation of executable applications makes the developed interface very flexible since it can be installed in many different computers without the need to have any version of LabVIEW installed.

5 Bibliography

[1] E. Dorronzoro, A. V. Medina, I. Gómez, J. A. Gómez and M. M. Monge, “A Standard-Based Body Sensor Network System Proposal,” in *IT Revolutions*, 2011, pp. 106-115.

[2] J. del Río, M. Martínez, D. Mihai Toma, A. Mànuel and H. G. Ramos, “IEEE 1451 HTTP server implementation for marine data,” E-prints UPC -Universitat Politecnica de Catalunya, 2011.

[3] M. A. Fernandes, S. G. Matos, E. Peres, C. R. Cunha, J. A. López, P. Ferreira, M. Reis and R. Morais, “A framework for wireless sensor networks management for precision viticulture and agriculture based on IEEE 1451 standard,” *Journal Computers and Electronics in Agriculture,* vol. 95, pp. 19-30, 2013.

[4] R. Oostdyk, M. C. T. and P. J.M., “A Kennedy Space Center Implementation of IEEE 1451 Networked Smart Sensors and Lessons Learned,” in *Aerospace Conference, IEEE*, 2006.

[5] W. Elmenreich and S. Pizek, “Smart Transducers - Principles, Communications, and Configuration,” in *IEEE International Conference on Intelligent Engineering Systems*, 2003.

[6] Y. S. Eugen and K. Lee, “Understanding IEEE 1451-Networked smart transducer interface standard,” *Instrumentation & Measurement Magazine, IEEE,* pp. 11-17, 2008.

[7] “IEEE Standards Association,” [Online]. Available: http://grouper.ieee.org/groups/1451/0/body%20frame_files/Family-of-1451_handout.htm. [Accessed June 2014].

[8] Measurement Computing, “Transducer Electronic Datasheet,” in *Data Acquisition Handbook*, 2012, pp. 125-127.

[9] IEEE Instrumentation and Measurement Society, *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats,* 2007.

[10] “Virtenio Official Website,” [Online]. Available: http://www.virtenio.com/en/products/radio-module.html. [Accessed June 2014].

[11] “The Contiki OS Website,” [Online]. Available: http://www.contiki-os.org/. [Accessed May 2014].

[12] “National Instruments Official Website,” [Online]. Available: http://www.ni.com/labview/. [Accessed June 2014].

[13] J. K. Rajiv, Digital Filters, Theory, Application and Design of Modern Filters, Wiley-VCH, 2012.

[14] National Instruments, Digital Filter Design Toolkit User Manual, 2005.

[15] F. J. Taylor, “Fixed-Point Effects,” in *Digital Filters, Principles and Applications with MATLAB*, Wiley, 2012, pp. 215-249.

[16] “Cortex Microcontroller Software Interface Standard (CMSIS),” [Online]. Available:

http://www.keil.com/pack/doc/CMSIS/DSP/html/group__group_filters.html. [Accessed July 2014].

[17] Texas Instruments, “TMS320C64x DSP Library Programmer's Reference,” p. Appendix A.

[18] “National Instruments Official Website,” [Online]. Available: http://www.ni.com/visa/. [Accessed 2014 May].

[19] T. Mandel, “The Golden Rules of User Interface Design,” in *The Elements of User Interface Design*, John Wiley & Sons, 1997, p. Chapter 5.

[20] “National Instruments Official Website,” [Online]. Available: http://www.ni.com/white-paper/3024/en/. [Accessed April 2014].