POLITECNICO DI TORINO

Corso di Laurea Magistrale in
Ingegneria Elettronica (Electronic Engineering)

Master's Degree Thesis

Realization of a Prototype for the Implementation of Non-Intrusive Load Monitoring

Thesis developed at Texas Instruments Inc.

Supervisor
Prof. Gianluca Setti

Candidate
Valerio Trobiani

Co-Supervisor
Daniele Cozzi

Academic Year 2020/2021

Abstract

The Non-Intrusive Load Monitoring is a technique that can be used to recognize appliances that are being used in a house from their electrical characteristics, such as the profile of the current that they absorb. This technique can be employed to potentially achieve an appliance by appliance complete disaggregation of the loads connected to a power grid and therefore of the ongoing energy consumption.
The Non-Intrusive Load Monitoring can have different applications, going from the partitioning of the electrical bill and the smart management of local power sources, like solar panels, to the *Ambient Assisted Living* for which we use it to check the normal continuation of the everyday life activity of the assisted person (e.g. an old person).

The purpose of the presented study is to realize a prototype which receive as input the waveform profile of the current absorbed from the power grid and that is capable of detecting the activation of a new appliance, called *event*, and of recognizing it from its characteristic waveform shape. Moreover, the prototype is intended to be capable of communicating the outcome of the performed classification to other devices via Wi-Fi by using an MQTT protocol, which is typically employed in IoT applications since it allows a short data transmission to multiple readers.

For the recognition mechanism we exploited a *Feed-Forward Neural Network* (FFNN) and we utilized as the identifying feature for each appliance the waveform profile of the current that it absorbs after a certain time from its activation, in its steady-state. More specifically, we observed the real and imaginary harmonic content of the acquired waveform up to the 16^{th}.

In particular, we used as the basic *Single Layer Perceptron* Neural Network structure, which is characterized by a single hidden layer and therefore by the lowest possible computational weight, which is particularly suitable for our application.

During the study, after an initial evaluation of the feasibility of the project in which we analyzed with an oscilloscope and a MATLAB modeling respectively the input noise and the minimum number of bits required in our ADC, we explored two different implementations of the described prototype: one in which the classification of the appliance is performed remotely in a cloud server and the other one in which it is carried out in edge, at local hardware level.

Both the architectures examined employ the same hardware resources that are an MSP430 reference design (TIDA-00929), which provides the signal chain for the integrated ADC, and a Cortex-M4 Launchpad (CC3220MODASF), which also mounts a module for wireless communication and an antenna that can be exploited for the MQTT-based Wi-Fi transmission required in the prototype. The communication among the two MCUs is established by mean of a UART channel.

In both the explored solutions, the prototype receives as input the profile of the current waveform that is absorbed from the power grid and detects the eventual activation of an appliance. Afterwards, in the case of the classification in cloud architecture it transmits via Wi-Fi an entire isolated section of the waveform acquired to a cloud server, where it is remotely performed the frequency analysis and the appliance recognition.

In the second architecture considered, instead, those same operations for the frequency analysis and the classification are done at the local hardware level and in particular by the Cortex-M4 processor.

At the end of the classification process, whether is was realized locally or in cloud, the outcomes are transmitted via Wi-Fi through an MQTT broker potentially to multiple other devices, such as smartphones. However, if we perform the recognition in edge we are able to directly communicate the result to other devices without the need of a remote cloud server.

The results obtained with these two solutions present a comparable accuracy in the appliances recognition, 92% for the classification in edge and 90% for the one in edge, but they strongly differ in the organization of the computational cost among the local hardware resources and the remote cloud, with the consequent need, in one case, of an appliance classification service in cloud and, in the other case, of a hardware powerful enough to perform the recognition.
The optimization of this computational distribution, as well as a study of the cost-effectiveness trade-off for the employment of hardware with different levels of performances, can be the foundations of further studies to be conducted on these subjects.

Contents

1 Introduction

1.1 Non-Intrusive Load Monitoring

The Non-Intrusive Load Monitoring, also called NILM, is a technique that is used to recognize appliances from their electrical characteristics.
The NILM is made possible by the fact that each appliance presents, in its electrical characteristics, a peculiar behavior, similar to a signature that identifies it and that can be exploited to build a system capable of recognizing the different types of devices.

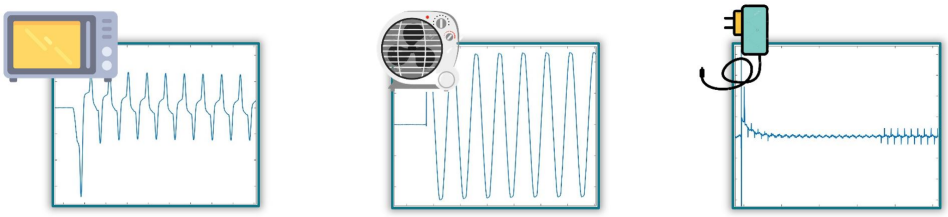


Figure 1: Every Appliance has its own (current) signature

The idea behind the Non-Intrusive Load Monitoring is to obtain a complete disaggregation, appliance by appliance, of the electrical loads connected to the power grid. This technique can have several applications, such as achieving a disaggregation of the electrical bill or a smart management of the local energy sources.

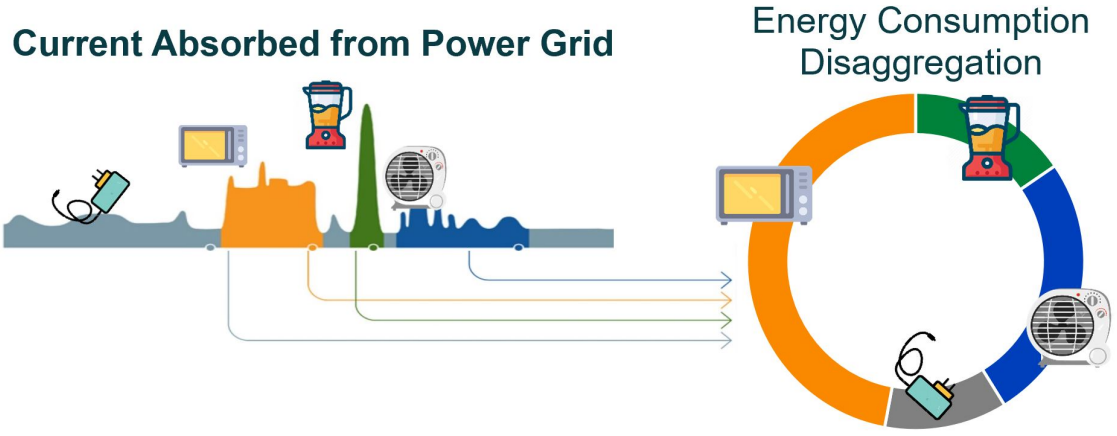


Figure 2: Example of Loads Disaggregation

The significance of this technique relies on its minimum intrusiveness, since the detection and the recognition does not happen where the power is actually consumed, at the power outlet level, like in ILM (Intrusive Load Monitoring), but it is performed at the source, so at the power grid level.

1.2 NILM advanced applications

The Non-Intrusive Load Monitoring can be exploited in several different applications besides the plain energy bill disaggregation. In fact, it can also be employed in more advanced uses, such as the Smart *Home Energy Management System* (HEMS) and the *Ambient Assisted Living* (AAL).

The former uses the Non-Intrusive Load Monitoring for managing efficiently the partitioning of the usage of the local power sources, such as solar panels, according to the punctual cost of the power consumed from the standard grid, the amount of power required and of the energy stored locally. With HEMS it is performed a smart choice, among others, of when to use the power coming from the grid and when to use the one produced locally.

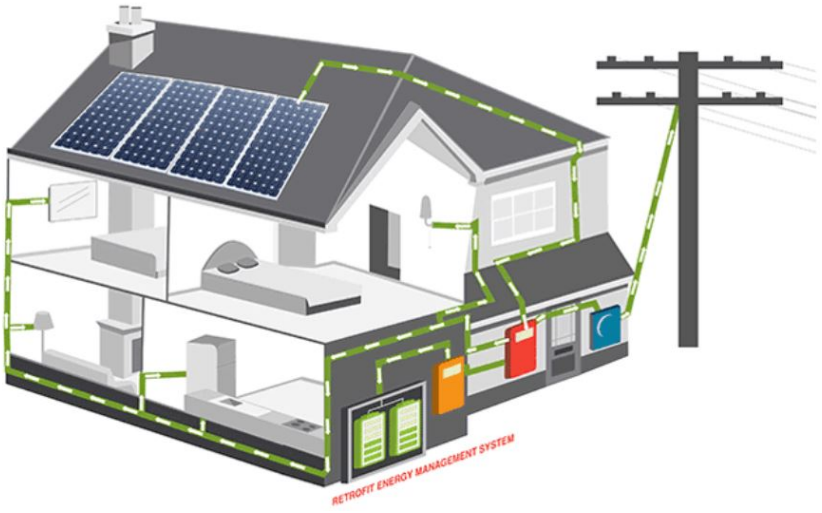


Figure 3: Smart Home Energy Management System

With the Ambient Assisted Living (AAL), instead, we can monitor the normal every-day life activities prosecution in the house of the person that we are monitoring. This application can be particularly useful in the case of elders, in order to be able to spot potential medical emergencies effectively and quickly and also to monitor, in general, the health state of the subject without any intrusiveness.

Figure 4: Ambient Assisted Living

1.2.1 NILM Realization Steps

Generally, in the realization of the NILM, we can identify a total of four steps, and each one of them will be conditioned in its characteristics and its effectiveness by the choices that we make. These steps are:

- Data Collection.
- Event Detection.
- Features Extraction.
- Load Classification.

Data Collection is the first step for the realization for the Non-Intrusive Load Monitoring, in which we need to select: what we want to measure; where we want to perform the measurements and the sampling frequency to use.

We need, indeed, to choose the electrical parameter(s) to measure in our devices. In particular, we can acquire, for example, the current profile absorbed by the appliances or their I/V trajectory and, from those, we can also compute and use the active/reactive dissipated power information.

Moreover, we need to select whether we want to perform the measurements at a single or multiple houses level and, finally, the sampling frequency to apply.

In particular, for the sampling frequency used for the NILM data collection, we can distinguish among different categories, going from the Very-Slow ones ($\leq$1Hz) up to the Extremely-High ones ($\geq$40 kHz).

The **Event Detection** step represents the capability of the system, that is implementing the Non-Intrusive Load Monitoring, of revealing the occurrence of an event, which is defined as the general activation (or deactivation) of an appliance, that has been connected to the power outlet or has been turned On (or Off).

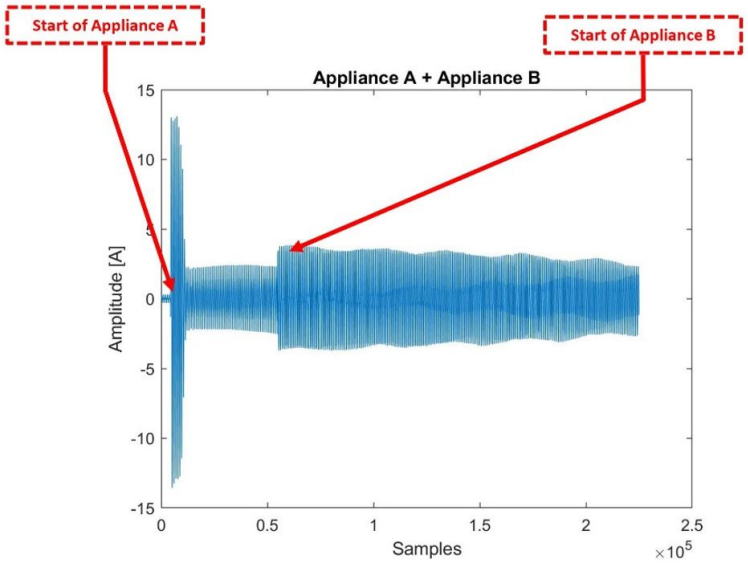


Figure 5: Example of Events on Current Measurements

The importance of this step relies on the fact that a good reliability of the event detection mechanism translates into an high repeatability of the performed measurements, and therefore to -an higher accuracy levels in the classification phase.

Features Extraction is the following step for the realization of the Non-Intrusive Load Monitoring and it consists in the extraction, from the electrical characteristics of the load that we have measured, of a proper feature that contains the signature of the appliance and that is coherent among the different measurements of the same device, in order to ensure repeatability in the signatures extraction.

At this level, we can select features that are either event-based, and therefore related to a measurement performed consequently to the occurrence of an event, or non-event based, hence a periodic measurement, from which it is generally harder to obtain a proper appliance recognition.
Furthermore, we can act on the transient of the event or on the steady state of the waveform, waiting for a certain time, after the event, before collecting the data.

Load Classification is the last step of the NILM. It is based on the coherent features previously selected and extracted and it is performed exploiting a recognition mechanism, such as machine learning.

At this point, it is worth mentioning that appliances can be divided into four different classes according to their general behavior and typical usage.

We can, in fact, have *On/Off* appliances, such as lamps or heaters, that provide a fairly constant behavior and whose activation lifetime can be fully determined by their being turned On- or plugged in a power outlet- and then turned Off- or unplugged.

Appliances can, otherwise, belong to the class called *Finite State Machine*, in which they follow, during their normal functioning, a cycle of different behaviors and perform a sequence of different actions. Examples of these appliances are washing machines and fridges.

We can, then, also have *Continuously Varying* appliances, such as working tools or dimmers, whose behavior can not be predicted a priori and, finally, we can have the *Permanent Consumers* type of appliances, like smoke detectors and security cameras, that are intended to be continuously active.

As it can be imagined, the appliances for which the realization of Non-Intrusive Load Monitoring results easier, and on which the following study has been based, are the ones belonging to the first class described, the *On/Off*.

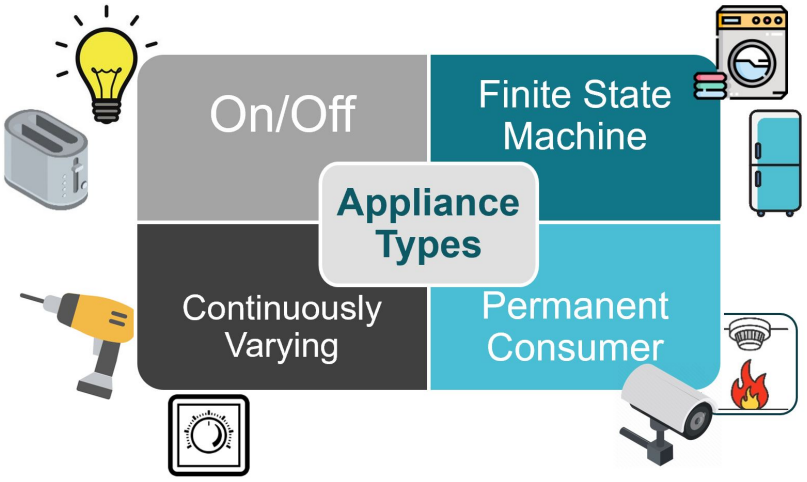


Figure 6: Classes of Appliances

1.3 Database Measurements

Since, in our study, we did not have the possibility of connecting probes to an actual power grid and measure personally the electrical characteristics of several appliances at their activation, we resort to a public database, called WHITED (Worldwide Household and Industry Transient Energy Data Set).

In the WHITED database there are the current and voltage profiles, during their activation phase, of 126 appliances. For each of them, the database provides a total of 10 measurements. The data present in this database were acquired with a 16-bits precision and a sampling frequency of 44.1 kHz.
Each measurement has a duration of 5 seconds and the actual activation of the appliance occurs after ~ 0.5 seconds.

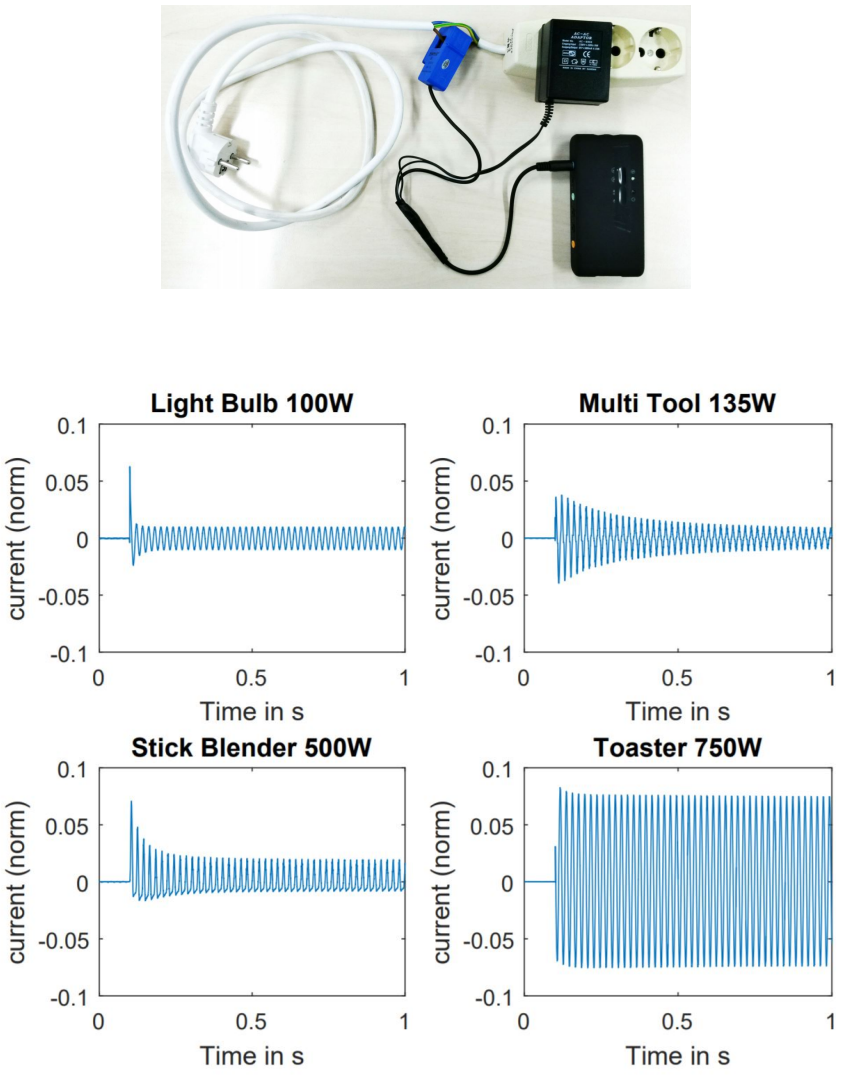


Figure 7: WHITED Database Setup and Measurements

1.4 Purpose of the Study

The purpose of this study is to realize a prototype which is capable of recognizing an appliance, that has been activated, from its electrical characteristics, such as the profile of the current that it absorbs from the power grid, exploiting the Non-Intrusive Load Monitoring technique described previously.

Furthermore, the prototype implemented, is intended to be capable through a Wi-Fi transmission of communicating the results of its processing to other devices such as, for instance, the provider's servers or the smartphone of the owner of the power grid.

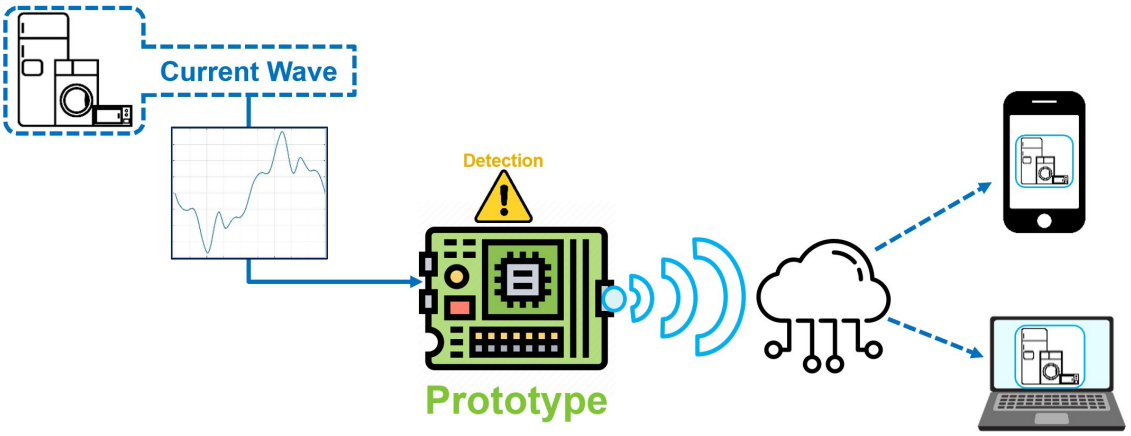


Figure 8: Project Overview

The first steps for the realization of this prototype regards the study of feasibility of the classification mechanism.

To do that, we exploit the Database Measurements previously described and a MATLAB modeling of the final behavior of the prototype and of the recognition system. The purpose of this process is to verify whether the obtained results are compliant with the possibility of actually realizing the intended device.

2 Prototype Theorizing

2.1 Prototypes Architectures

During the study, we explored two different solutions for the realization of the generic described prototype with the same overall architecture and hardware resources but with very different behaviors. In fact, the first step is to analyse the case of a prototype in which the actual classification process is performed in Cloud, exploiting the higher computational power available, and from which the results can be distributed among other devices, such as smartphones.

The second case of study is the one of a prototype in which, instead, the classification is realized directly in edge, at the local hardware level, and where we only use a Wi-Fi communication to transmit the result of the recognition to other devices.

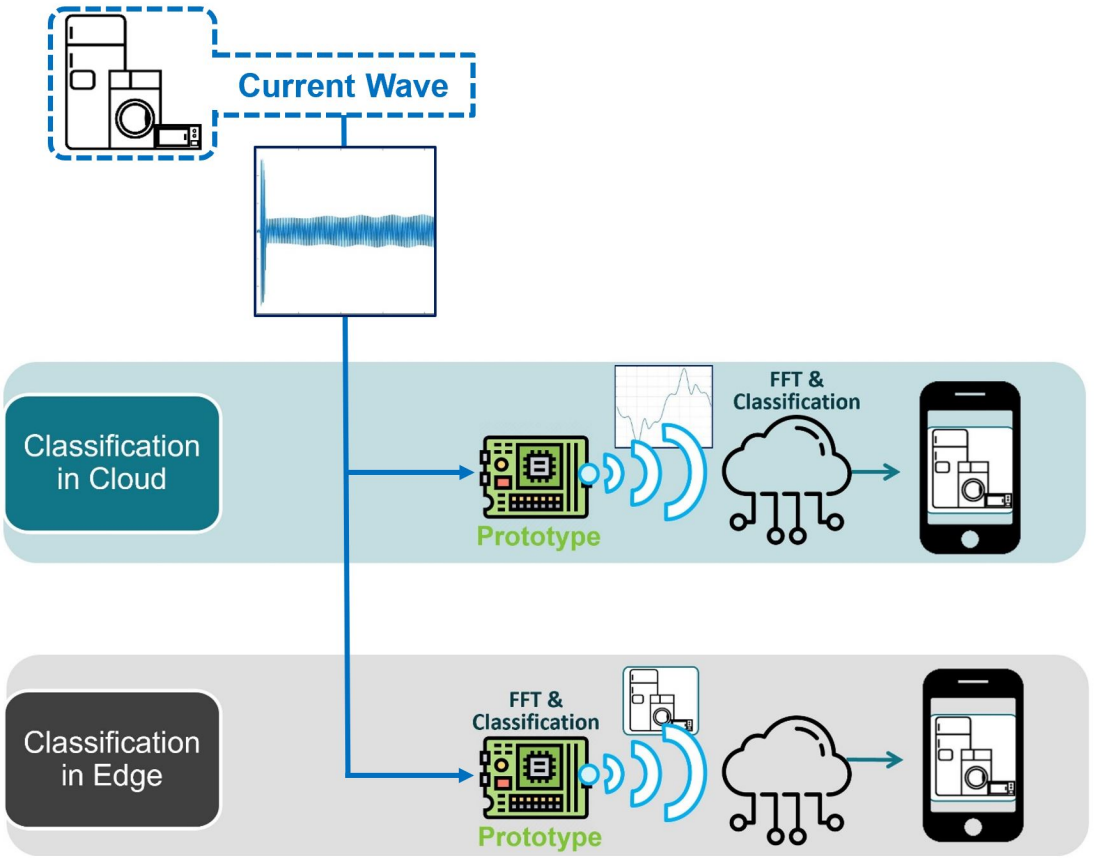


Figure 9: Prototypes Comparison

In Figure 9 it's shown that in the case of *Classification in Cloud*, the prototype receives as input to its ADC Signal Chain the entire current waveform and after its acquisition it performs from that the event detection (identification of the activation of a new appliance) and the isolation of a section of the signal in its steady-state. Afterwards, it transmits via Wi-Fi to a cloud server the entire isolated signal and where it is remotely performed, at cloud level, the actual processing for the appliance recognition.

In the second case, the one of *Classification in Edge*, we still receive as input the same current waveform signal, and we still perform the operations of event detection and signal isolation just described but, in addition, we also realize the processing for the appliance recognition at the local hardware level. After that, we still exploit a Wi-Fi communication system to transmit the final result of the classification, which is distributed among other devices. Therefore, in this case, no actual operation is performed at cloud level.

2.2 Starting Point of the Project

The project realized and here presented had, as a starting point, a previous study conducted on the topic and, in particular, the following aspects of the system had already been explored:

- **Hardware:** ADC Signal Chain
- **Software:** Database values normalization
- **Firmware:** Early Algorithm for the Event Detection

2.3 Appliances Signature Feature Selection

The feature that we exploit for the appliance recognition mechanism, in which therefore must be contained a signature of the appliance that is consistent among different measurements and which differentiates each device, is contained in the spectrum of the steady-state waveform of the current absorbed by the appliance. In particular, we detected the presence of a signature feature in the odd harmonics extracted from the signal analysed and we used them up to the 16^{th} harmonic.

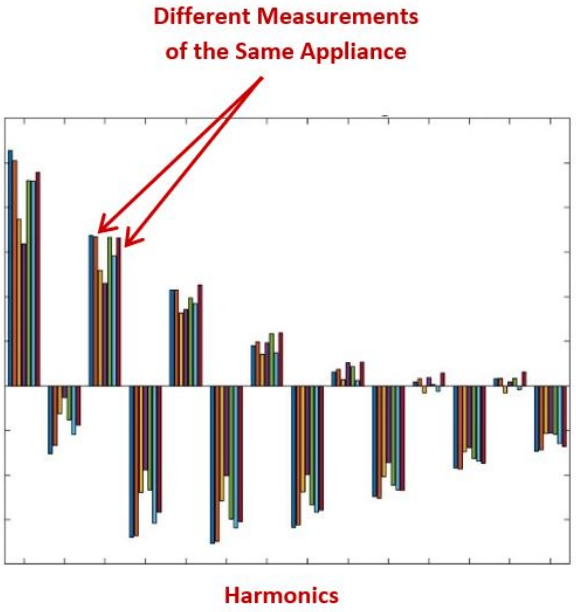


Figure 10: Appliances Signature Feature

3 Neural Network for Appliances Recognition

3.1 Recognition Mechanism

The recognition mechanism to be chosen for the purpose of our study must allow proper classification performances, with a good level of accuracy, but it must also provide the lowest possible computational effort, since one of the purposes of the research is to be able to perform those operations at local hardware level.

For those reasons, the recognition mechanism employed is a *Feed-Forward Neural Network* (FFNN) with the structure of a *Perceptron*. It accepts as input the values of the harmonic content of the isolated signal and return in the output the index of the corresponding classified appliance. In particular, the perceptron structure will ensure a minimum computational effort, since it is only made of Hidden Layers placed between the Input and Output ones and therefore it will reduce the complexity and the cost of the hardware required for the network to be recreated into it.

3.2 Data-Set Generation with Matlab

During the Matlab modeling of the problem, we need to consider that most of the operations that we are examining will be performed at MCU level, and so they need to be kept as simple and straightforward as possible when modelled with Matlab, without making full use of the huge computational power of the tool. In fact, otherwise it would be fairly impossible to perform similar operation with the MCU, and our Matlab analysis would result to be meaningless.

In the analysed case of appliances recognition starting from their current measurements, we need to extract the harmonics content from the current samples and, in particular, to isolate the odd harmonics in the real and the imaginary spectrum.

The first thing to do is to convert the two-channels .flac files, provided by the WHITED database, into raw current samples, using the Matlab function *audioread*(). Then we need to apply a correction factor, assigned by the Database, according to the region in which that particular measurement was taken.

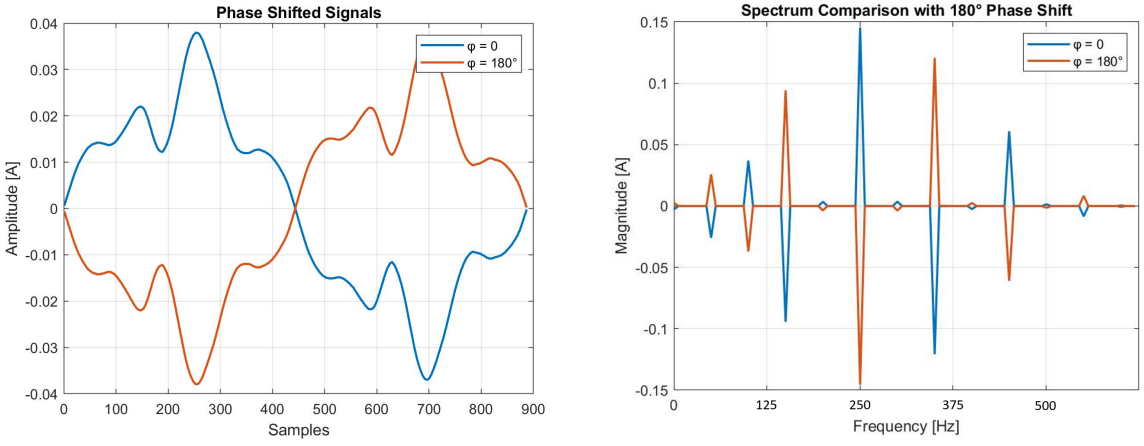
After that, we need to isolate a section from each current measurements, to resemble the limited memory capacity available at Hardware Level.

3.2.1 Algorithms for Single Period Isolation

1. At this point, the purpose is to isolate a single period of the current wave considered. A first attempt to pursue this goal is done using a Zero-Crossing function, which points out where the data crosses the zero-value. This function can be useful to try to highlight a single wave period by using its three consecutive zero-crossing values $[ZeroCrossing(3) - ZeroCrossing(1)]$. To obtain more consistent data, we consider the measurements after the initial transient, which lasts on average $0.2sec$, and so, considered the database Fs (44.1 kHz), after $\sim$ 20000 samples, starting from the event detection, we will surely be in the steady state condition.

However, this first attempt, presents some issues that would lead to a poor overall recognition performances. First of all, the signal may present some fluctuations around the zero, making the Zero-Crossing function strongly unreliable. In addition, we have that, using this algorithm, we may end up isolating, for the different measurements of same appliance, periods that are in phase opposition.

Due to the fact that we are considering a single period of the original wave and the separate real and imaginary spectra, this phase shift within an appliance measurements, results into an erroneous extraction of the harmonic content in the isolated periods, for which, with a phase inversion, their real and imaginary harmonics are detected as inverted in sign, leading to an impossibility of a coherent training of the Neural Network and of achieving consistent classification results. This effect can be appreciated in Figure 11, where are depicted the spectrum of two identical isolated periods with a phase inversion between them.



(a) Signals Comparison

(b) Real Spectrum Comparison

Figure 11: Phase inversion effect on spectra

To solve the problem of wrong periods extractions for those signals that present fluctuations around zero, we exploit the fact that we are aware, within a certain error range, of the value of the fundamental harmonic ($\sim$ 50 Hz), so we can take, as the end point of the single wavelength, the next zero-crossing after $\sim 1/50sec$. Being our sample frequency Fs, equal to $44.1kHz$, this corresponds to an 880 samples shift.

Although we are now able to isolate a single period of the current waves, we are still not able to guarantee a phase coherence between the periods extracted from the different measurements of a single appliance. To solve this problem, a first attempt is made by verifying, by looking at the sign of the mean of the firsts 100 values after the first Zero-Crossing detected and, in case of a negative average we would multiply the isolated period by -1 to flip it.

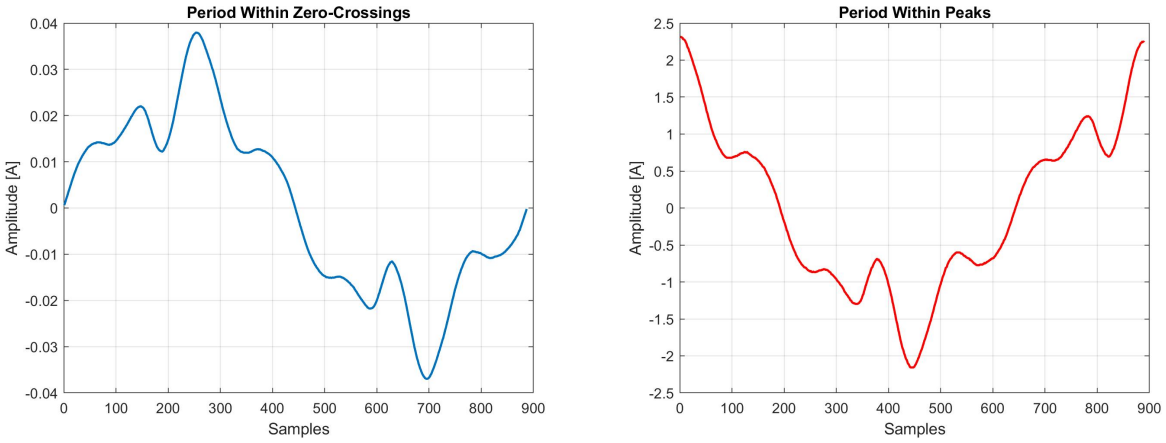
Anyhow, since this would alter the actual waveform, this period isolation algorithm is abandoned.

2. The second algorithm evaluated to achieve a coherent isolation of a single period from the current wave, consists in detecting the peaks of the signal in order to isolate a period as the wave in between two consecutive local maximum.

The main issue with this algorithm is that the signals very often presents several fluctuations that can lead to an erroneous detection of the peaks. As before, we consider the wave in its steady state, which means $\sim$ 20000 samples after the event detection. The starting peak of the period can be easily computed exploiting the awareness, withing a certain error range, of the fundamental harmonic value. In fact, we can compute the maximum of the signal within the 880 samples, because we know that the fundamental is at $\sim$ 50 Hz and so within that range we will surely have an entire period.

For the second peak detection, we can't assume that its value will be consistent with the previous one due to the high irregularity of the measured current waves. Thus we can use the same approach as before by taking the maximum of the signal within the 880 samples after the first peak.

With this algorithm we are able to perfectly isolate in-phase signals and the periods extracted start and end with peaks as we can see in Figure 12.



(a) Zero-Crossing Algorithm

(b) Peaks Detection Algorithm

Figure 12: Different isolation techniques outcomes

3. To properly isolate coherent periods from the different measurements of the same appliance in terms of phase and still being able to obtain a signal that starts and ends at the zero-point, we resort to a mixed approach, that combines both concepts exploited in the previous two algorithms.

In fact, we use a peak detection mechanism, similar to the one used in the second proposed algorithm, to point out two subsequent peaks, again in the steady state of the wave, that, as previously stated, will identify a complete period of the signal. Hence, we are now perfectly aware of the exact number of samples that compose the analysed period. Now we can exploit the Zero-Crossing function, described for the first algorithm, to find the first zero of the signal that occurs before the first detected peak and then we isolate the period by using the exact number of samples that compose it previously retrieved. The procedure is depicted in Figures 13 and 14.

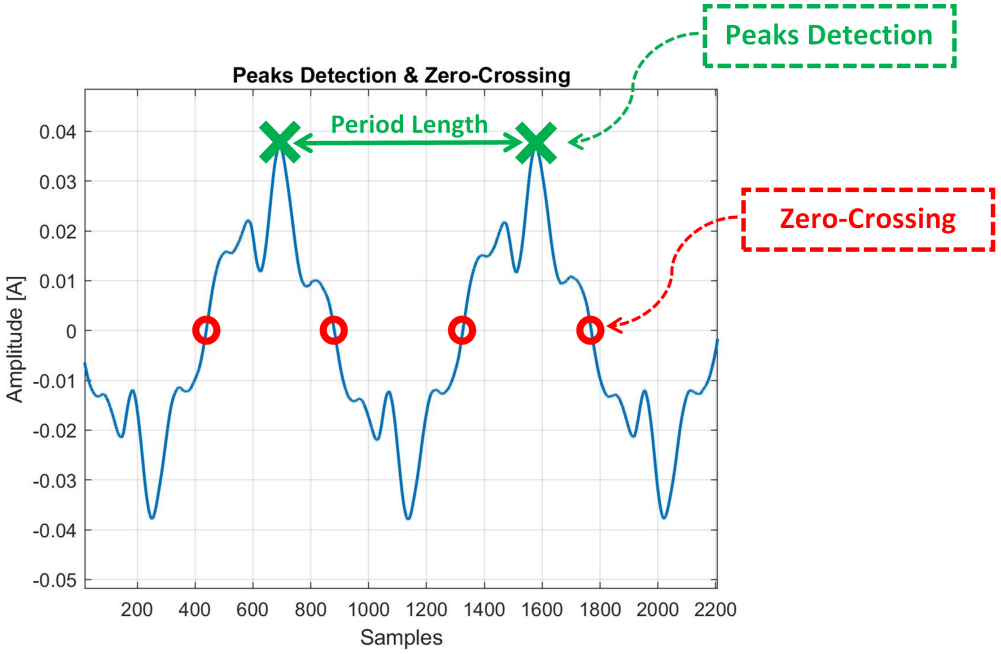


Figure 13: Mixed Algorithm: Step 1

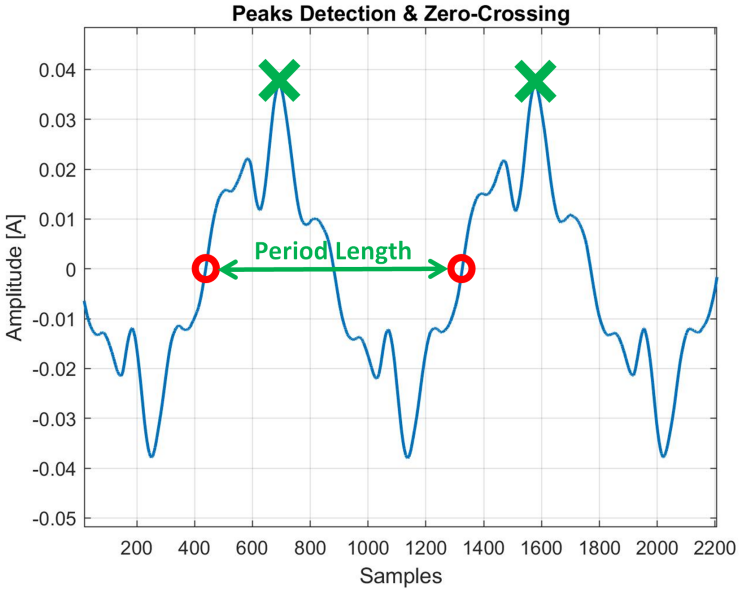


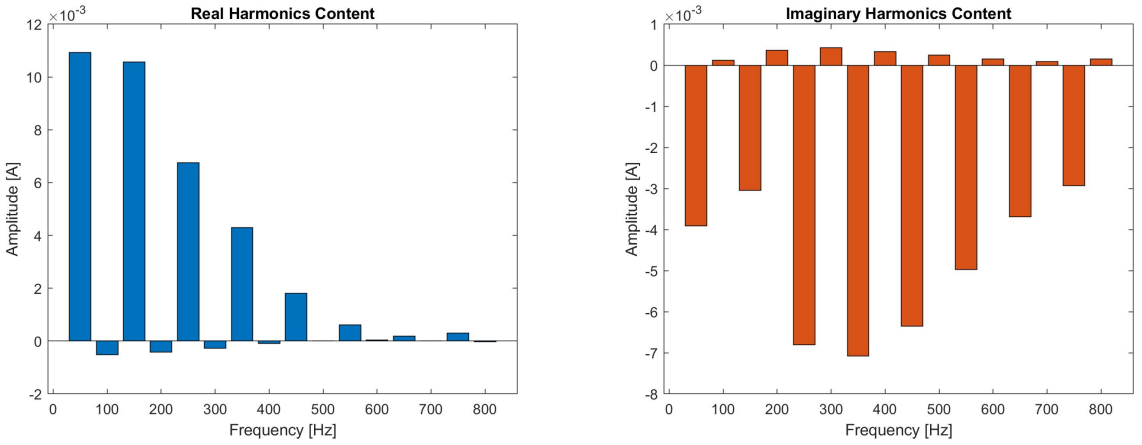
Figure 14: Mixed Algorithm: Step 2

3.2.2 Features Extraction

Once the current wave period has been isolated, in order to obtain a more accurate spectrum content, we extend each wave by repeating the extracted single period for a total of eight times.

After that, we can proceed with the Fast Fourier Transform and the separation of the Real and Imaginary parts of the spectrum. Knowing, within a certain error range, the value of the fundamental harmonic, we are able to detect it precisely using the $max()/min()$ functions and similarly we are able to detect all the higher order harmonics.

Finally we need to isolate the Odd ones, for both the Real and Imaginary spectrum, and to aggregate them into a single vector, obtaining, for 126 appliances with 10 measurements each, a total of 1260 vectors of Odd harmonics values. In particular, each vector is made of 14 elements and it is constructed as $[1_{RE}$; 1_{IM} ; 3_{RE} ; 3_{IM} ; 5_{RE} ; 5_{IM} ; 7_{RE} ; 7_{IM} ...].



(a) Real Harmonics Content

(b) Imaginary Harmonics Content

Figure 15: Separate spectra

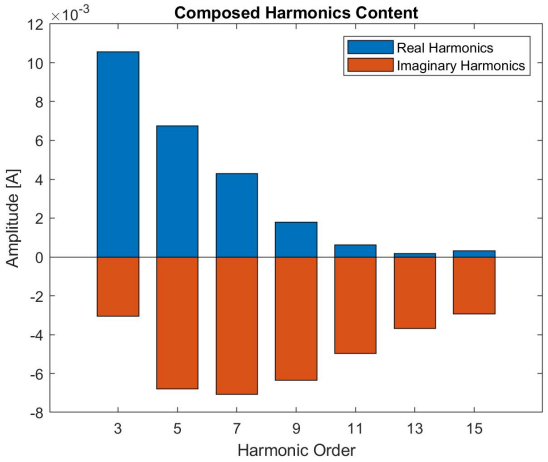


Figure 16: Composed Harmonics Content

3.3 Neural Network Characteristics

As mentioned previously, our choice for the implementation of a *Feed-Forward Neural Network* capable of classifying current waves from their spectral content, among a total of 126 classes, falls on the use of a simple *Perceptron*. To be able to identify a fairly high number of classes, like in this case, we need to select accordingly the number of Hidden Layers and the number of neurons in each of them.

However, it is not a trivial task, since if we make our Neural Network too complex, with an excessive number of layers and neurons with respect to the number of input and output neurons, our Neural Network will be susceptible to bias when trained, and it will not be able to accomplish the generalization principle, falling in the case of *overfitting*.

Whereas, if we select an insufficient number of hidden-layer neurons, our network will not be able to approximate our model, and we will not have accurate outcomes, falling in an *underfitting* case.

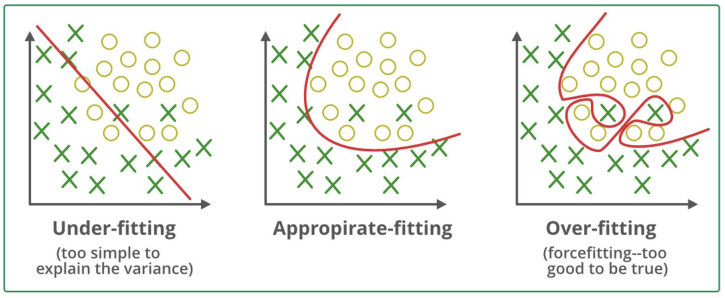


Figure 17: Recognition Issues

For these reasons, we use a single hidden layer for our Network and we select the number of neurons in it by exploiting the general rule of thumb for which:

$$N^o\ H.L.\ Neurons = \frac{N^o\ Samples\ in\ Training\ Dataset}{\alpha\,(N^o\ Input\ Neurons \times\ N^o\ Output\ Neurons)} \qquad \alpha \in (2;5)$$

Therefore, the *Feed-Forward Neural Network* chosen will have a structure similar to the one in Figure 18, corresponding to a *Single-Layer Perceptron*.

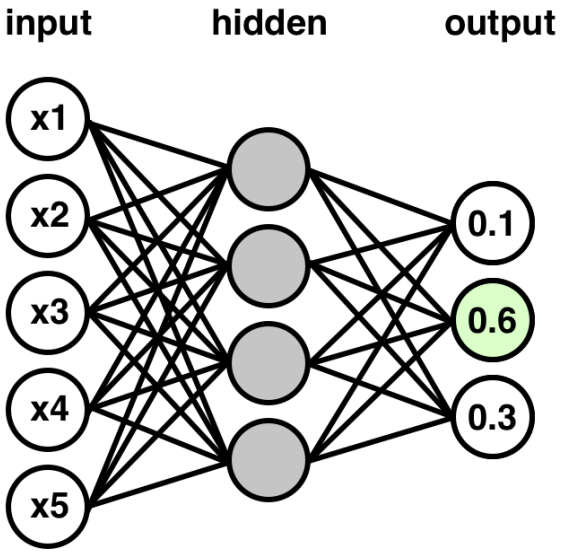


Figure 18: Neural Network Visualization

3.4 Neural Network Training

To train effectively the Neural Network (NN) with a Supervised technique, which provides higher accuracy in this case, we need a vast training data-set of harmonics vectors, extracted from the current waveform provided by the database, with the corresponding correct classification outcomes. However, we want to avoid the occurrence of overfitting in the training data set and so, in order to have enough training data, without introducing a bias in the NN, we remove, for each appliance set of 10 measurements, 3 of the 10 harmonics vectors extracted, and we will use those only to measure the accuracy of the NN and not for its actual training.

Then, we use the 7 remaining vectors to extend our data-set generating a total of 50 vectors for each appliance, without introducing a bias. In fact, we build these 50 vectors by using a random noise signal, with a variance equal to the one measured among the starting 7 harmonics, and overlapping this noise with the actual harmonics values from those 7 vectors. Moreover, to further reduce risk of a biased training, the harmonics with the same order, belonging to different vectors of the same appliance, are mixed randomly. From the Figures 19 and 20 below it is possible to notice how, from the 7 starting odd-harmonics graphs, the several vectors created present coherent characteristics, even though they are all different from the original ones and from each others.

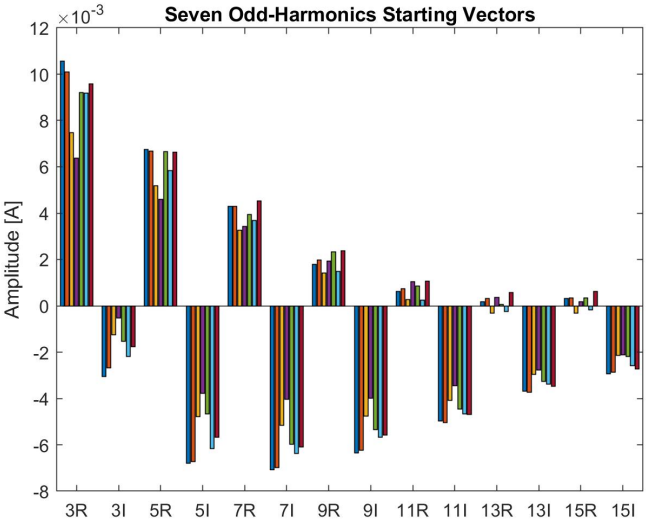


Figure 19: Starting Seven Odd-Harmonics Graphs

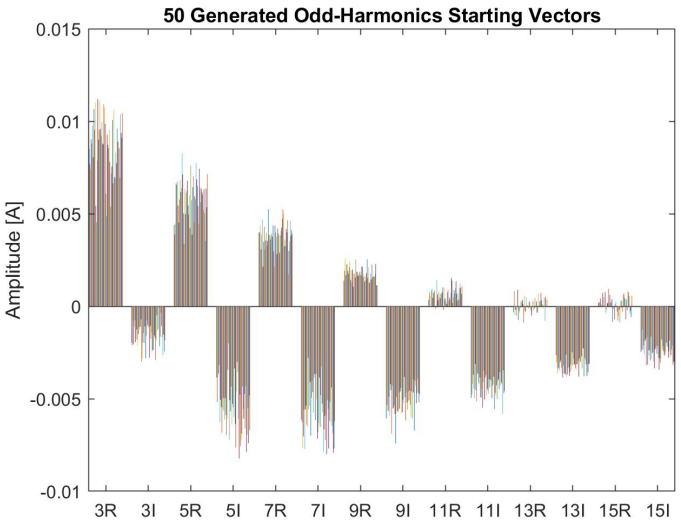


Figure 20: 50 Generated Odd-Harmonics Vectors

3.5 Partial Results and Problems

Following the steps so far discussed, we are able to generate and train a Neural Network capable of recognising and classify the 126 appliances. However, when we perform an accuracy analysis with the three Odd-Harmonics vectors previously removed from the data-set, we are only able to achieve an accuracy of $\sim$ 85% - 88%, which is not acceptable, considering that we are performing a Matlab modeling, while these operations will actually be performed at MCU level with a considerable worsening of the measurements.

3.6 Data-Set Reduction

Analysing in details the results of the performed accuracy measurement, we can easily learn that the overall performances of the NN are not equally distributed among all the different considered appliances. In fact, we can point out some of them for which, due to the characteristics of their current waves, the followed steps lead to a very poor accuracy outcome.

To avoid the increase of complexity in the computational weight to be handled by the MCU, instead of making the algorithm more complex and precise, we identify those appliances that are less likely to be correctly identified, and we remove them from the the data set.

To recognize these problematic appliances, we build a Matlab script which computes the accuracy of the generated Neural Network by feeding it with the different measurements provided for each appliance, we monitor the output of the Network and compares it with the expected outcome, verifying whether we have an Hit or a Miss. The built script, in parallel with the standard computation of the overall performance of the network, also keeps track of the disaggregate accuracy related to each single appliance.

In this way we can easily recognize those appliances that poorly perform due to their characteristics. We can notice, for example, the presence of two appliances that, due to their small current amplitude and to the presence of a slight offset, never cross the zero value in proximity of the event, causing the algorithm to be unusable without a pre-processing of the measured data, e.g. shift of the signal mean to zero. An example of these appliance current wave is reported in Figure 21.

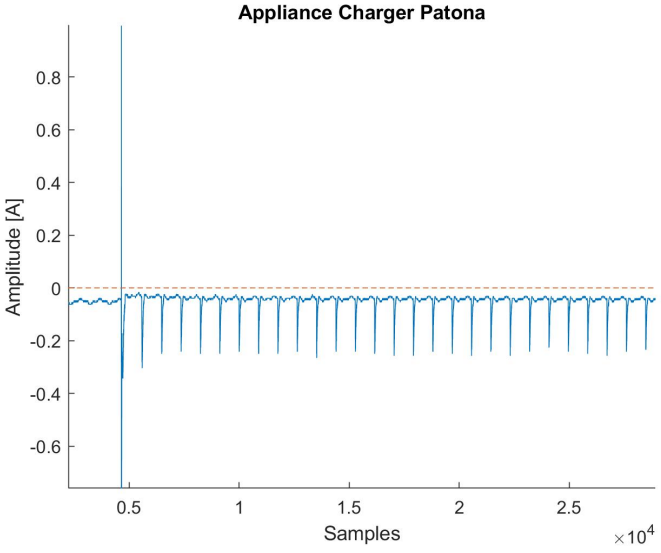
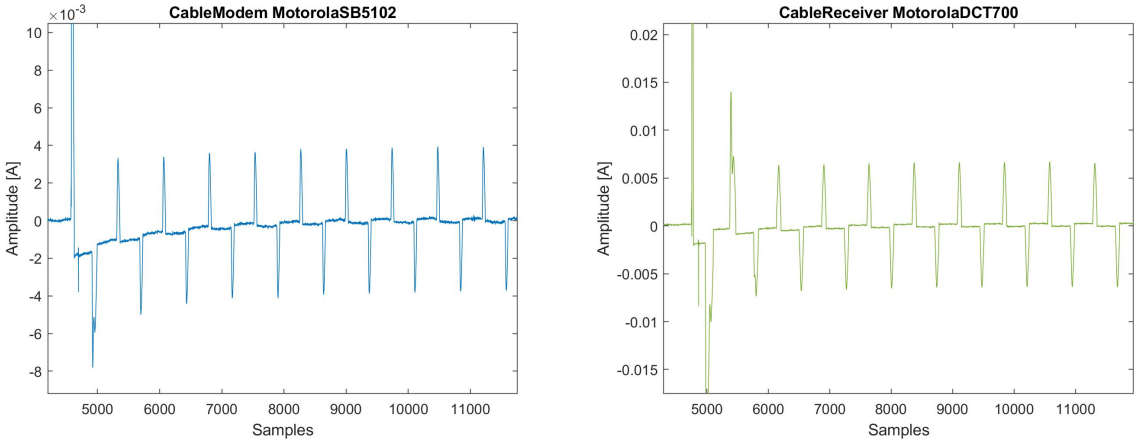


Figure 21: Non Zero-Crossing Appliance

Additionally, we discover some appliances that do not provide enough coherence within their different measurements and along their steady state to let us achieve an acceptable recognition performance and some others that present a current trend which is too similar to other appliances already listed in the database, leading to frequent classification errors, as depicted in Figure 22.



(a) First Appliance

(b) Second Appliance

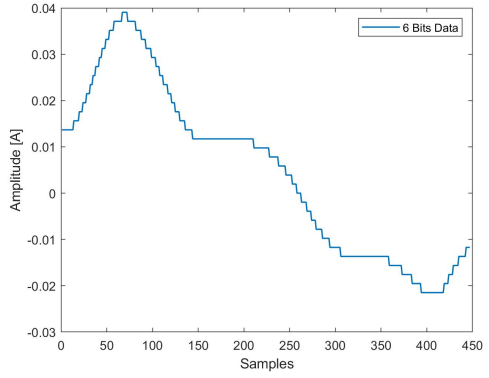
Figure 22: Different Appliances with Similar Current Waves

Finally, we can remove from the original data-set all the appliances that can't be recognized with an acceptable accuracy by the algorithm proposed, in order to significantly boost up the performance of the Neural Network. We identify 21 problematic appliances, on a total of 126, which is a fairly small percentage that still leave us with a notable number of elements in the data-set. Consequently, the reduction of the data-set is considerable largely acceptable, hence we lower the number of recognisable appliances to 105, obtaining an increase in the accuracy resulting from 83% up to $\sim 95\%$.

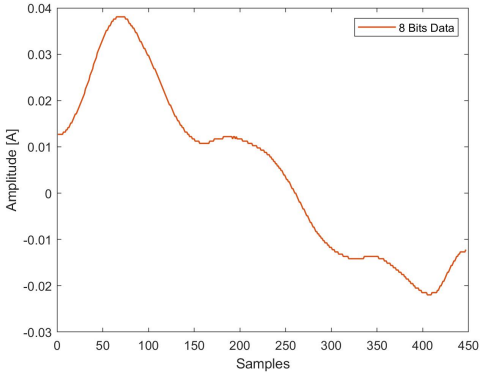
3.7 Fixed Point Analysis

The Fixed Point Analysis is the verification of how the accuracy of our network changes by modifying the number of bits used to represent data.

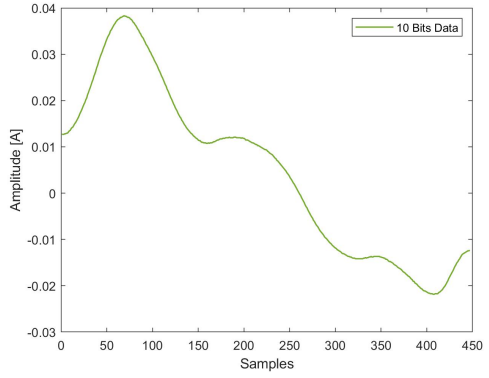
In fact, the analysis conducted so far has considered the ideal case of Floating Point numbers and operations, which allows us to reach a level of precision that goes beyond the one that we will actually be able to achieve with our ADC and MCUs. By reducing the number of bits we have a loss of information in the values of the measurements and also at the occurrence of each operation, due to the final performed rounding.



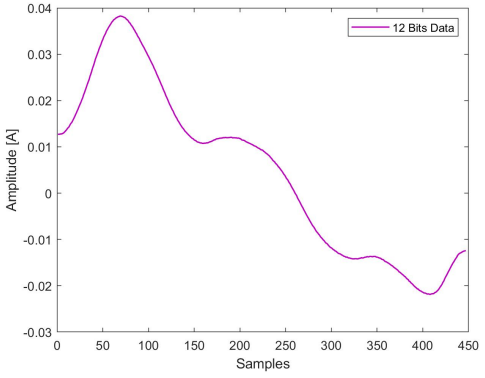
(a) 6 Bits Data



(b) 8 Bits Data



(c) 10 Bits Data



(d) 12 Bits Data

Figure 23: Loss of Information for lower bits conversions

Moreover, the fixed point analysis is fundamental to highlight which is the minimum *Effective Number Of Bits* (ENOB) that the Data Converter that we will use in our Prototype should have in order to keep the accuracy of the classifications similar to the one retrieved in Floating Point.

It results that, to achieve an accuracy higher than 90%, the number of bits must be at least equal to 10, as shown in the following table.

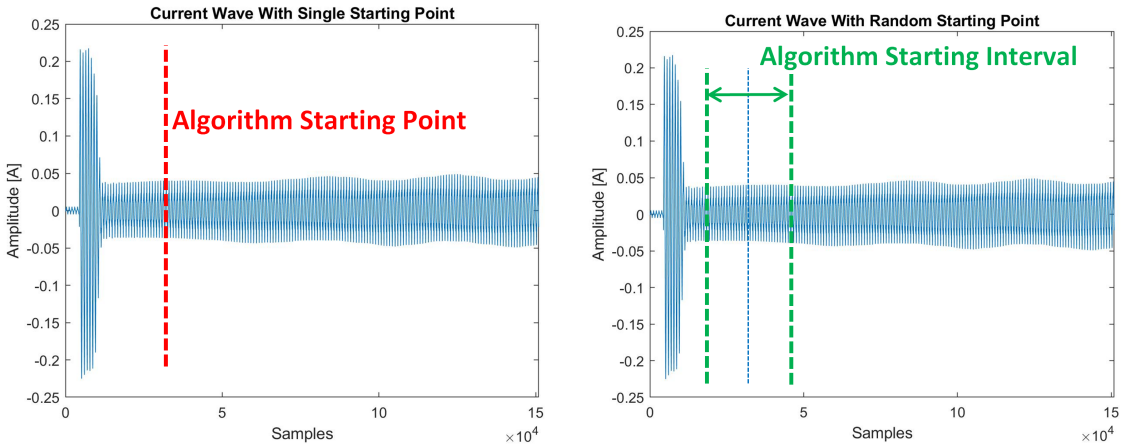
Fixed Point Analysis
Peak Detection & Zero-Crossing

Number of Bits	**Accuracy**
7	$< 70\%$
8	86%
9	89%
10	93%
≥ 11	94%

Table 1: Fixed Point accuracy results

3.8 Randomization of the Starting Point

During the analysis performed so far, the waves from which we isolated a single period and retrieved the spectrum content were considered and processed always starting from the same point, in the steady state. However, we need to take into account the fact that, in our final prototype, it will be virtually impossible to start to apply the period-isolation algorithm after an exact amount of samples with respect to the occurrence of the Event Detection. Thus we introduce an uncertainty of $\pm 0.25 sec$, which corresponds roughly to $\pm 10000 samples$, to the starting point for the application of our algorithm and we monitor how the performance of the Neural Network changes, also according to the number of bits used.



(a) Fixed Algorithm Starting Point (b) Randomized Algorithm Starting Point

Figure 24: Starting Point Techniques

From the obtained results, which are considerably close to the one previously retrieved, for the different number of bits considered, we can conclude that the proposed algorithm is robust against fluctuations of the starting point of the wave at which it is applied. The Neural Network is, however, retrained with the signals obtained with the described random starting-point. The results are presented in Table 2.

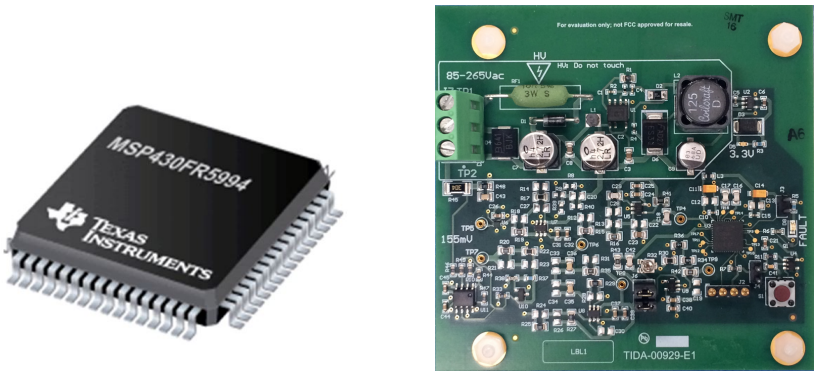
Fixed Point Analysis Peak Detection & Zero-Crossing Random Start Measuring Point	
Number of Bits	**Accuracy**
7	$< 70\%$
8	85%
9	87%
10	93%
≥ 11	94%

Table 2: Fixed Point accuracy results with random starting point

4 Micro-Controller Setup

4.1 Hardware Equipment

The Micro-Controller used for this study is the MSP430FR5994, mounted on the Reference Design TIDA-00929.



(a) MSP430FR5994

(b) Evaluation Board TIDA-00929

Figure 25: μC and Evaluation Board

This μC provides a 12-bits ADC based on an SAR architecture (Successive Approximations Register) which ensures a good level of conversion speed, achieving 200 Ksps, potentially being able to generate a conversion output after a number of clock cycles equal to the number of bits of the input, and a low complexity level, exploiting a single comparator, a DAC and a Successive Approximation Register. Moreover, its small complexity level translates into a fairly low power consumption.

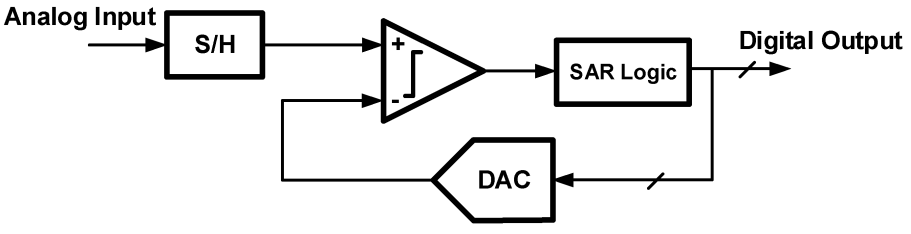


Figure 26: SAR-Based ADC architecture

The MSP430FR5994 is a Micro-Controller whose main purpose is the signal processing, hence it provides also a Low-Energy Accelerator (LEA) module, to support and speed up vector-based arithmetic and, in general, signal processing. Therefore,

it is capable of accelerating complex operations on signals, such as filtering, by mean of digital FIR filters, and the Fast Fourier Transform. For the purpose of our study, those capabilities are strictly needed, therefore this explains the choices made on this Micro-Controller as the one to be used.

The TIDA-00929, on the other hand, provides, among other things, a protection circuit for the ADC input, to avoid the failure of the device for an input that exceeds the MCU limits; a signal conditioning chain to minimize the SNR_q (Signal over Quantization Noise) due to input signals that do not exploit the entire input dynamic of the ADC available; and a JTAG interface to easily be able to program the μC using an external USB-Debug-Interface.

Figure 27: USB-Debug-Interface

The purpose of the study is to build a working prototype of an appliance event detector and recogniser, so, in order to test test our device, instead of actually connecting the Analog ADC input of the board to an eMeter to measure the current waves produced by the powering on of different appliances, we connect its Analog input to the audio source of a Laptop, by using an audio Jack with exposed wires, as shown in Figure 28, and we transmit through it the current wave signals retrieved from the WHITED Database.

Figure 28: Audio Jack with exposed wires

In order to transmit through the audio output the signals from the database representing current waves, we need to convert them, using once again Matlab, into .wav playable and single-channel audio files. This is realized using the *audiowrite*() Matlab function.

4.2 Measurement Noise Evaluation

To verify the actual feasibility of the project with the proposed setup, the first step is to evaluate the noise level introduced at the input of TIDA' s signal chain, so at the end of the audio cable, by performing the transmission of the signal from the audio Laptop source, through the audio cable. We connect the audio jack to the Laptop and the exposed wires on the other end of the audio cable directly to the Oscilloscope, to measure the signal that will be seen at the input of our prototype. In this way we are able to estimate the noise level introduced by the transmission system and judge, approximately, whether this mechanism is consistent or needs to be modified.

4.2.1 Long Wire Transmission

An initial attempt is performed by using a fairly long audio wire (∼ 80cm).

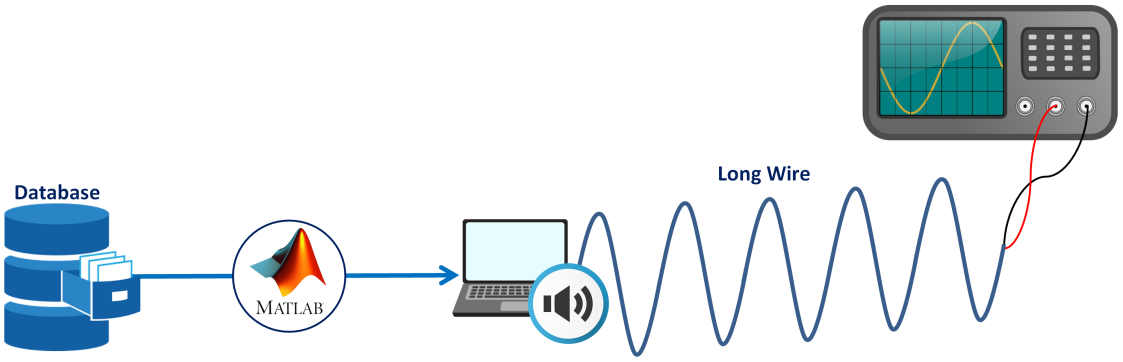
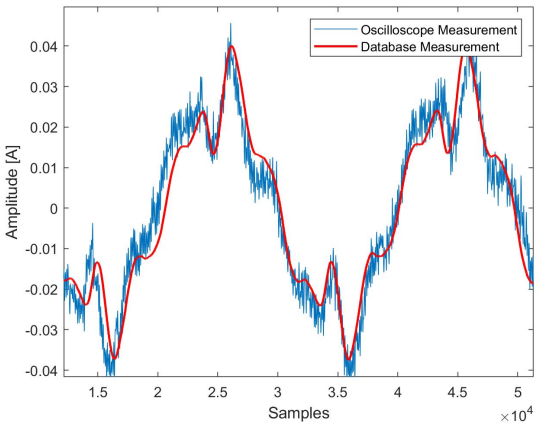


Figure 29: Transmission with Long Wire

Analysing the results retrieved from the Oscilloscope with this transmission configuration, we notice an heavy presence of noise, introduced likely by the use of such a long audio cable, as we can see in Figure 30, where it is compared with the original wave from the Database.

(a) Oscilloscope Measurement



(b) Oscilloscope and Database Comparison

Figure 30: Long Wire Measurements

However, to be sure that the signals transmitted in this way are unusable and unreliable, we save the waveform captured by the Oscilloscope, we transfer the data collected into our PC and we feed it to our Neural Network in MATLAB, which gives us wrong classifications, confirming the need of modifying the data transmission mechanism.

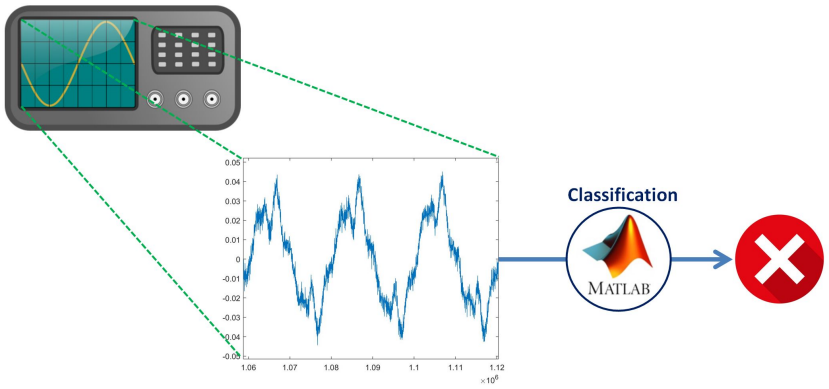


Figure 31: Wrong classification of extracted period

4.2.2 Short Wire Transmission

At this point, we make the hypothesis that the encountered noise can be due to the length of the wire through which the audio signals are transmitted.

Therefore, we proceed removing most of the cable, reducing its length to exactly 7cm.

Now we connect, once again, the audio Jack to a Laptop and we reproduce with it the .wav audio file obtained from the conversion of the current wave signal. Also this time, we measure the waveform at the exposed part of the cable with an Oscilloscope

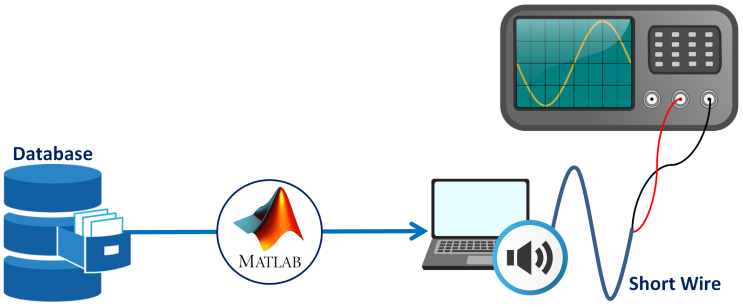


Figure 32: Short Wire measurement

This time, the measurements retrieved with the oscilloscope present a much cleaner profile, with a significant lower level of noise. In fact, after performing a simple decimation of the measured data, we obtain the wave reported in Figure 33.

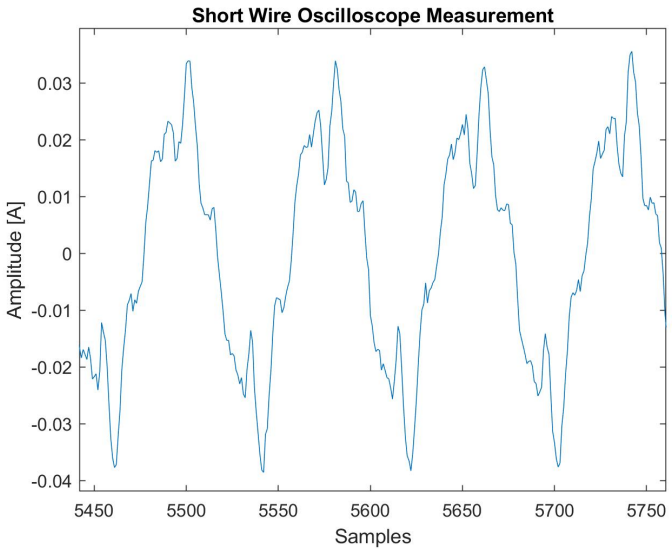


Figure 33: Decimated Oscilloscope Measure with Short Wire

Now we can proceed, similarly to the previous case, feeding the wave extracted from the Oscilloscope to the Neural Network in MATLAB and this time we obtain correct classifications.

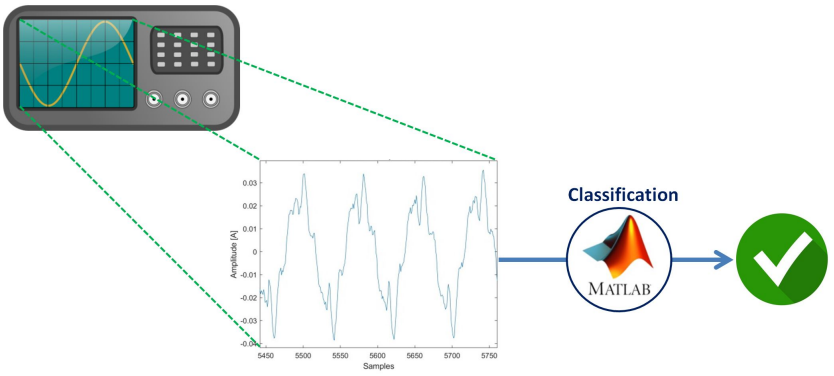


Figure 34: Right classification of extracted period

This suggests that the initial hypothesis that led us to a reduction of the cable length, was correct and that, with the proper modifications applied, the transmission mechanism tested can be considered valid and can be used for the further analysis conducted on our prototype. Therefore, the shortened audio cable is soldered on the Evaluation Module, as depicted in Figure 35.

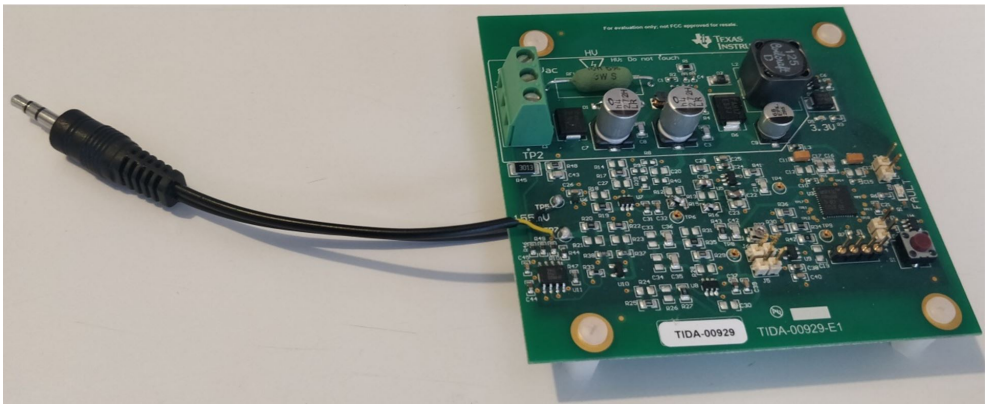


Figure 35: EVM with Short Wire

4.3 Transmission to Micro-Controller

The last test is to verify the adequacy of the transmission mechanism, is to stream the data from the Laptop to the board ADC input, and check the sampled and converted data directly from the MCU memory.

To do that, we program, using the JTAG interface of the TIDA and the external USB Debug Interface, the Micro-Controller, with the Event-Detection algorithm, which has already been developed in the previous study conducted on this subject.

The Event-Detection is capable, by exploiting the 12 Bits SAR-Based ADC of the μC and a Band-Pass FIR filter, to detect the occurrence of an event, which consists in the powering on of an appliance, by reading the relative current wave absorbed. Therefore, we run this program and, as soon as an event has been detected, we interrupt it, leaving the circular buffer of the MCU, in which the ADC stores its acquisitions, filled with valid signal waves.

Hence, by using the USB Debug Interface and the Software tool Code Composer Studio, we are able to extract the signal values from the Circular Buffer of the MCU and verify their quality, to properly analyse whether our data transmission mechanism and the SAR ADC provided are adequate.

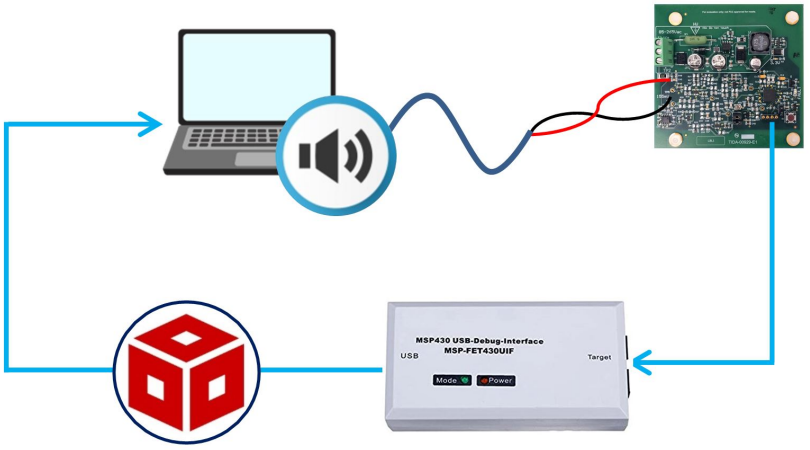


Figure 36: MCU Transmission Validity Test

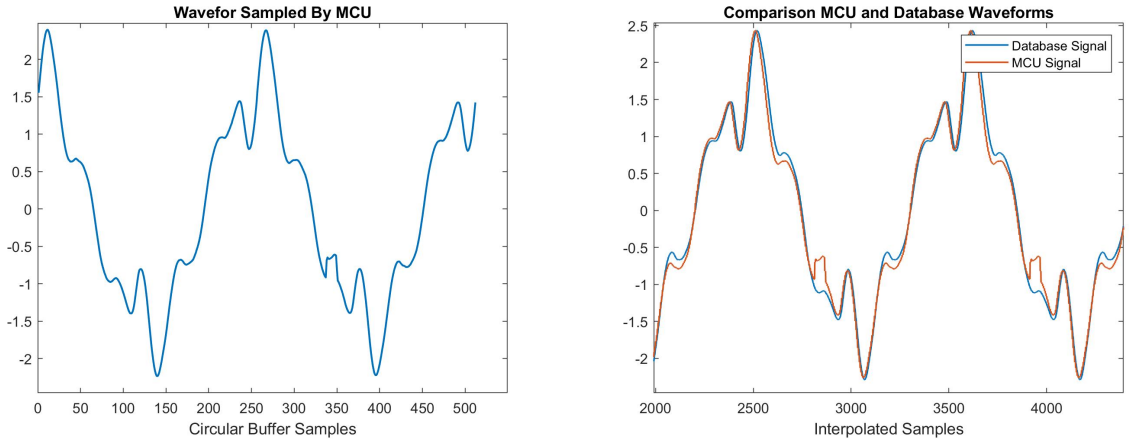
Since the MCU, in our test program, samples at $12800Hz$ and the Circular Buffer can store 512 Words, being aware of the fundamental frequency of the signal $\sim$ 50Hz, we expect to find in it exactly 2 periods, as demonstrated below:

$$Samples\ Per\ Period = \frac{Fs}{f_0} = \frac{12800Hz}{50Hz} = 256$$

$$\downarrow$$

$$N^o\ Periods\ In\ Buffer = \frac{Buffer\ Length}{Samples\ Per\ Period} = \frac{512}{216} = 2$$

Therefore, we are able to collect the periods of the transmitted waveform from the MCU Circular Buffer. From the data retrieved in this way, presented in Figure 37, we can notice that, as predicted, are present exactly two waveform periods and, in addition, we can observe a strong similarity between the wave extracted from the MCU and the ideal one from the Database.



(a) MCU Sampled Waveform

(b) Comparison with Database Waveform

Figure 37: MCU Circular Buffer Content after Event Detection

Finally, we isolate a single period from the waveform collected from the MCU Circular Buffer and we feed it into the Neural Network and we obtain a correct Classification Outcome. This ultimately confirms the adequacy of the transmission and measurement system built and, as a side effect, also proves the robustness of the generated Neural Network.

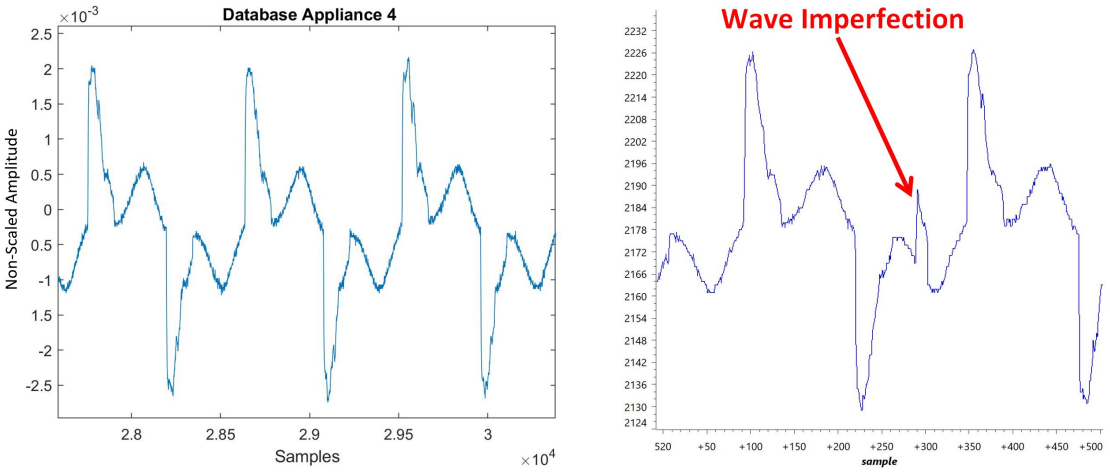
Moreover, the quality observed in the signal sampled and converted by the MCU and extracted from the Circular Buffer, may lead us to confirm, in the final prototype, the use of the integrated SAR ADC, abandoning any hypothesis of substituting it with an higher precision external one. Using the integrated ADC, together with a reduction in the project complexity, would also contain significantly the final Bill of Material (BOM) and, therefore, the cost of the final product, making it more competitive on the market.

4.4 Micro-Controller Wave Sampling Tests

To verify the consistency of the measurements performed by the Micro-Controller, we try to transmit, again with the Laptop audio player, through the standard audio Jack, different current waves taken from the database and opportunely converted. Once again, after the event detection, we can check the content of the Circular Buffer by interrupting the program running on the MCU.

In order to make this operation simpler and to avoid the insertion of breaking points in the code, we want to transfer, after the Event Detection, the waveform contained in the Circular Buffer into another memory location, using another variable, that we call Wave Memory. In this way the desired waveform can be monitored from this additional variable at any moment, while, instead, the Circular Buffer's content is continuously modified.

At this point, while testing other current waves, we notice that, due to the mechanism used by the ADC and the MCU to save the data samples into the Circular Buffer, for which it is considered as made of two separate halves that are filled one after the other, it is present, in the overall signal, an imperfection, which corresponds to the point in which the two Circular Buffer halves are logically connected. This effect is due to a carrier frequency of the signal which is slightly different from the ideal 50Hz one, and can be easily spotted in the Figure 38.



(a) Database Waveform (b) Circular Buffer Data

Figure 38: Circular Buffer Data Imperfection

The circular buffer has been defined, on purpose, with a length of 512 Words, so that a wave period, that we know having their fundamental frequency at 50Hz, sampled at 12800Hz by the MCU, will occupy exactly half of the Circular Buffer Length, hence 256 Words.

Therefore, we can avoid the occurrence in our Wave Memory of the imperfection found in the Circular Buffer, by copying in the new location, initially, only one half of the Circular Buffer, which will contain an entire period of the signal, using the function *memcpy*(). Afterwards, we duplicate in the remaining half of the Wave Memory its own content, obtaining a total of two contiguous wave periods, without any imperfections between them, as can be appreciated in Figure 39.

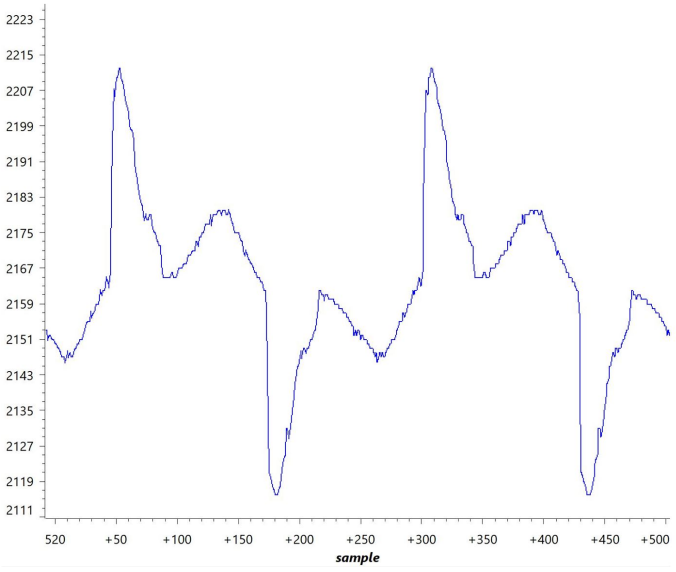
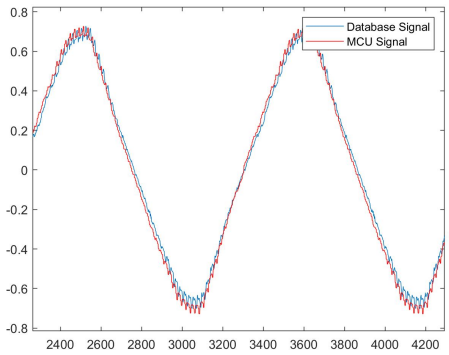
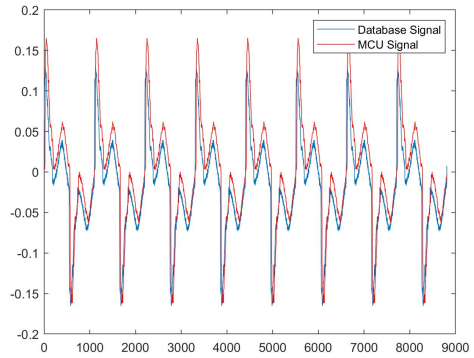


Figure 39: Wave Memory Data

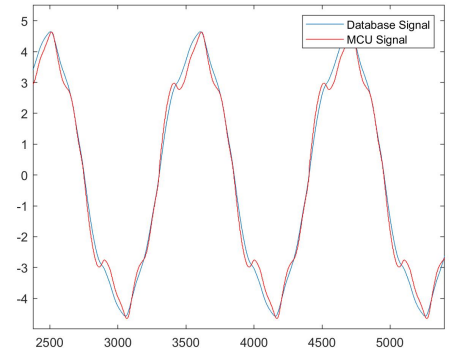
Now we proceed analysing other appliances, verifying every time whether the MCU is able to correctly sample the input waveform and if the Number of Bits provided by the on-board SAR ADC allow us to obtain a correct classification. In the following graphs are reported, for four appliances (namely the number 2, 4, 18 and 41) randomly picked, the comparisons between the Original Database waveform and the signal retrieved from the MCU Wave Memory.



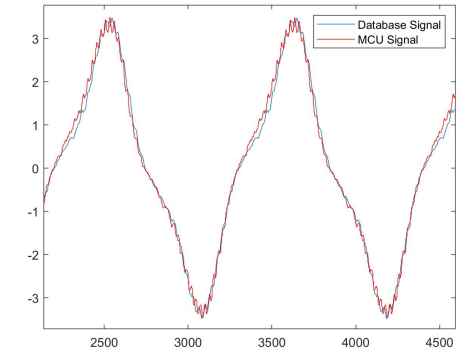
(a) Appliance 2 Database Measurement 1



(b) Appliance 4 Database Measurement 1



(c) Appliance 18 Database Measurement 1



(d) Appliance 41 Database Measurement 1

Figure 40: Comparisons Between Database and MCU Signals

From the tests conducted with different appliances, characterized by heterogeneous waveforms, we can conclude that the event detection mechanism works properly in identifying the occurrence of an event, e.g. an appliance being turned on.

However, we also notice that, extracting the sampled and converted waveforms from the MCU's Wave Memory and feeding them into our Neural Network, we obtain inconsistent results, with a fairly high number of misclassifications. At this point, we make the hypothesis that this can be due to the relatively small number of ENOB provided by the on-board SAR ADC (~ 10).

In fact, as we can see, for example, in Figure 40, appliances with waveforms with a very specific signature, like the (a.) or (d.) can be recognized by the Neural Network also with an evident presence of an high Quantization Noise, while those appliances, like the (c.), that provide a much less characterized waveform, we need a good level of precision in the MCU measurements to be able to distinguish them from the other similar ones.

This is true also for those appliances that, even though they have a well characterized signature, they share a similar behaviour with some other ones, making the recognition in need of a good level of precision and therefore, possibly, an higher number of bits.

In any case, from the tests performed, the level of precision of the classifications applied to the waveforms extracted from the MCU, settles around 65-75%.

However, before deciding whether we need to substitute the currently used ADC with a more accurate one, we will need to perform an accuracy analysis using, as training data-set of the Neural Network, actual waveforms extracted from the MCU, in order to train the Network with more realistic inputs, since the loss in accuracy may be due to an inadequacy of the Neural Network itself.

4.4.1 MCU Data Classification Issues Solution

In the previous steps of the study, we have highlighted how, signals acquired from the MCU and then extracted from it, are not classified effectively, obtaining an accuracy of $\sim$ 65% - 75%.

This can be due to the fact that the analysis performed on Matlab, on the Database signals, even though it was conducted in fixed point, relied on signals that are intrinsically different from the one that we can extract from the MCU, since in the latter ones there will be a different Noise Floor, Jitter Noise and Quantization Noise.

To be able to achieve a good accuracy with this kind of waves, we need to change completely the Data-set involved in the Neural Network Training mechanism. This is fundamental, because at this moment, we are trying to classify MCU-Acquired waves with a Neural Network that has been trained with Database signals.

Therefore, we try modifying the training data-set for our Network by substituting the database signals with the actual signals extracted from the MSP430 memory that previously we were not able to classify effectively.

The actual neural network training data-set was generated exactly as discussed

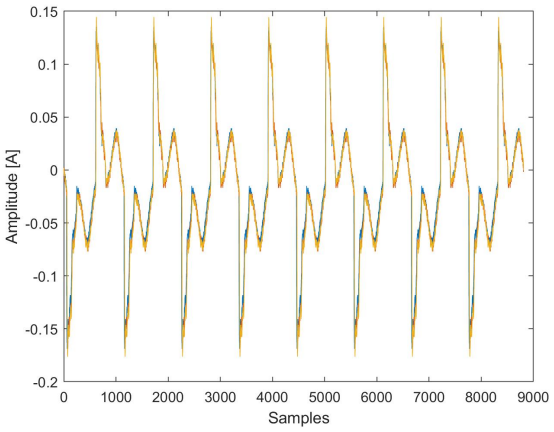


Figure 41: Waveform extracted from MCU and database comparison

previously, creating 50 waves per appliance from 7 of the 10 signals, using a randomization of their values within the actual variance that exists between them. However, in this case, the 10 starting signals, were the ones extracted directly from the MCU memory.

We can now proceed performing the classification, after verifying the coherence of the training executed by classifying the remaining three signals extracted from the MCU acquirements, and we obtain a fairly high accuracy ($> 90\%$) for the appliances under test. From this analysis we learn that we are indeed able to classify with high accuracy the signals that we extract from the Micro-Controller Memory if we train the Neural Network with real case on-field waves and not with the Database ones.

5 UART Communication

Considering that our prototype shall be able to perform a transmission of data via Wi-Fi, we will need to resort to an additional board, since it is not possible to realise a wireless communication system with the hardware so far presented, as the TIDA-00929 does not provide such a capability.

As a consequence, we will need some additional hardware to implement the communication with the cloud and, moreover, we need to implement a communication mechanism to transmit the data acquired and processed by the TIDA-00929 to the new additional board, still to be defined and chosen.

Since, indeed, the additional board, to which at this point we can refer to as the WiFi-Board, has not been chosen yet, we select a communication mechanism between that and the TIDA, which is extremely common and eventually relatively easy to implement with standard IO pins. Therefore, we choose the UART (Universal Asynchronous Receiver-Transmitter). In this way, once we have correctly implemented and tested the UART transmission from the TIDA, the WiFi-Board that we will select will most likely have such an interface and will be able to communicate with the TIDA.

5.0.1 UART Protocol

The UART is a single-master single-slave asynchronous serial communication protocol, which means that it allows the communication only between two ICs at the time. It can be designed as Simplex, Half-Duplex or Full-Duplex, according to whether, respectively, only one of the two connected ICs can write on the line; both of them can write on it (not simultaneously) or there are two separate transmission lines for the two ICs to write and read at the same time.

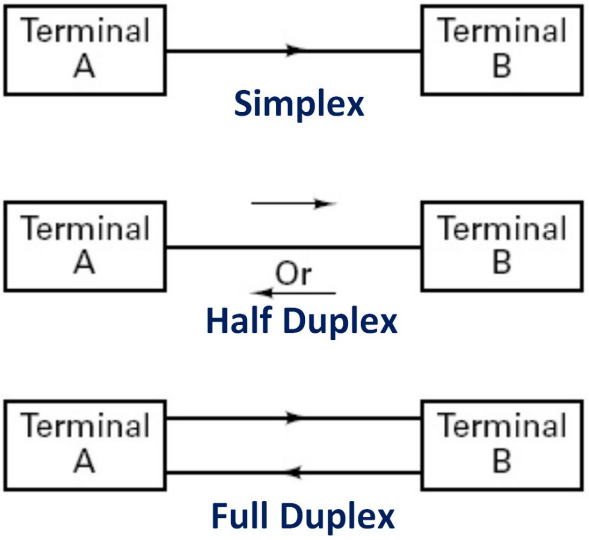


Figure 42: UART configurations

The idle condition of the UART transmission line is the $high - bit$ state and the start of the communication is indicated with a Start bit, that, of course, to be distinguished by the idle condition will be a $low - bit$.

After the Start Bit, a total of 8 bits (1 Byte) are transmitted and then the communication is closed with a stop bit, which makes the line go back to the idle high condition.

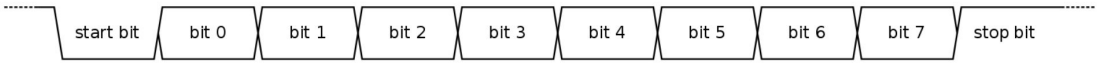


Figure 43: UART protocol

The UART protocol is also characterized by a Baud-Rate, which must be the same between the two connected ICs. For the purposes of this project we will communicate using a full duplex UART protocol, which is supported by the Hardware available so far, and a standard Baude-Rate equal to 9600.

5.0.2 TIDA UART Testing Hardware Setup

To implement and easily test the C code for the UART communication, we use at this initial stage another board which mounts the same Micro-Processor as the TIDA-00929, so that the code will be perfectly reusable on the TIDA, but that provides a much larger GPIO interface, which makes the debugging much easier. The board chosen is the LaunchPad MSP-EXP430FR5994.

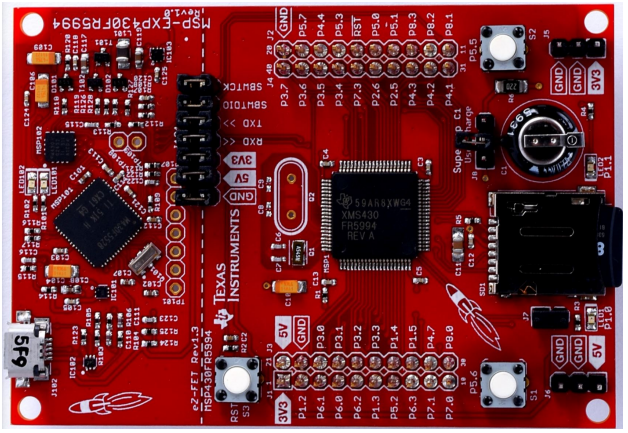


Figure 44: LaunchPad MSP-EXP430FR5994

The use of this additional board is due to the fact that the TIDA-00929, even though the MSP430 has a Full-Duplex UART channel, does not provide the related GPIO pins. In fact, on the TIDA, we can access the UART interface of the MSP430 only through some Testing Points (TP), which are much harder to access than standard GPIO Pins and, for most of them, it would be also required a soldering of an external jumper for testing and debugging, as can be seen from the zoomed TIDA detail reported in Figure 45. The UART testing point on the TIDA would be TP9 (for TX) and TP10 (for RX).

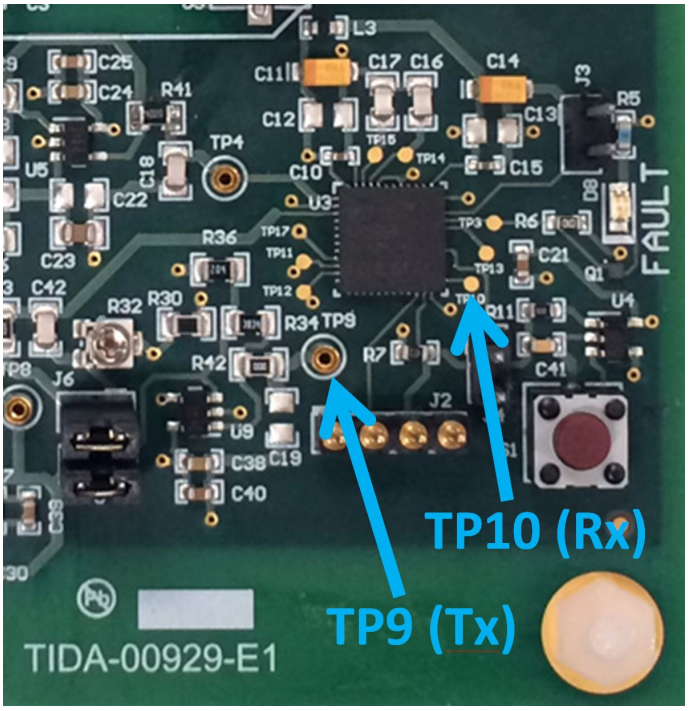


Figure 45: UART Testing Points on TIDA

Therefore, we can now proceed by designing a proper C code for the MSP430FR5994 processor, that will be compatible with both the MSP-EXP430FR5994 and the TIDA-00929, for the UART communication, since they both mount the same Micro-Controller.

5.1 UART Implementation and Debugging

The purpose, at this point, is to verify the correct functioning of the UART interface of the MSP430 and of our code. Hence, we exploit the USB debugging interface that is present on the Launchpad.

1) LaunchPad → LaunchPad

The first logical step to achieve an effective TIDA → LaunchPad communication is to design a code, for the latter one, capable of transmitting data from its UART TX port and reading those same data from its RX port, shorting the two pins with a jumper. Thus, at this first stage, we are only using the LaunchPad, as a debugging platform.

In order to be able to verify separately the functioning of the TX and RX sides of the code, we can check whether the board is actually transmitting by connecting its UART Transmission port to an Oscilloscope.

In this way we are able to tell whether, in case of a non-functioning code, the problem is in the TX or RX part of it. The setup is shown in Figure 46.

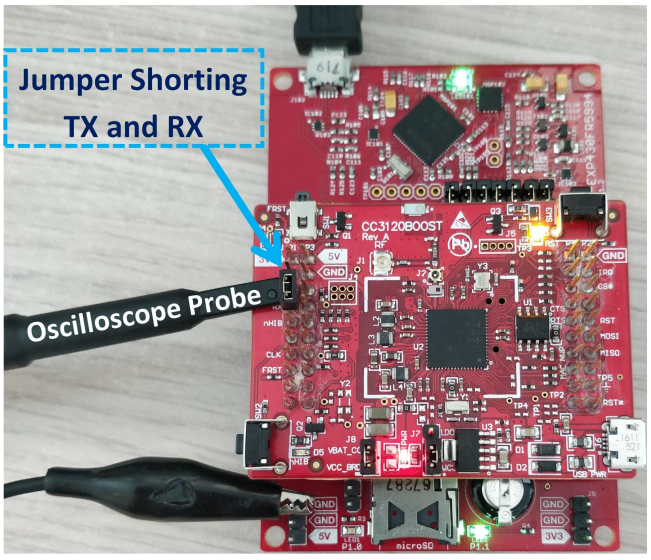


Figure 46: Shorted TX-RX Testing Setup

After designing the UART transmission code and completing the debugging we are able to collect, with the oscilloscope, a valid UART transmission, which is also detected and correctly read by the Board itself from its RX port.

The signal collected with the Oscilloscope is reported in Figure 48. All the C codes can be found in the Appendix 10.1.

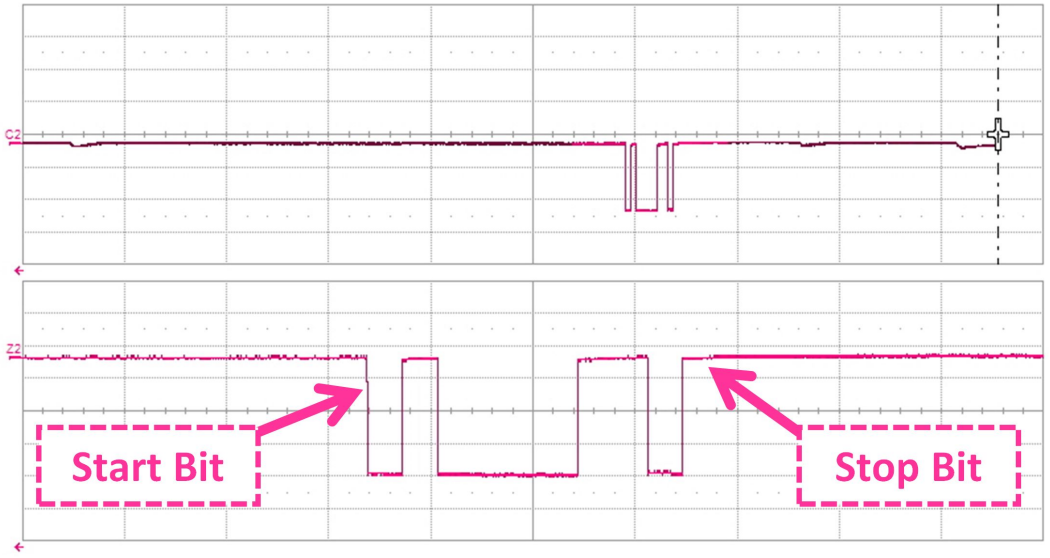


Figure 47: UART Transmission Detected with Oscilloscope

2) LaunchPad → PC

The next step is to transmit the data to our PC through the USB port of the board, still using only the LaunchPad MSP-EXP430FR5994 and the UART protocol. In fact, another advantage that the LaunchPad presents with respect to the TIDA is the possibility of debugging and communicating with the PC through the same USB interface which is used for power supply without any external Debugger MSP-FET.

The capability of communicating with the PC through the USB port will be useful for the entire project for debugging and to collect much faster the current wave signals from the MCU that we will use for the training of the Neural Network. However, this communication mechanism will not be actually present in the final prototype.

To perform the data transmission through the USB channel we modify the code used in the previous step, by enabling the LaunchPad USB UART Full Duplex Channels. To test its functioning, we initially transmit through it some known data while in the PC we read the corresponding serial port using the software Tera-Term. Similarly, we are also able to read the serial port with Matlab, by using the functions *serialport*() and *read*(), which will be useful when we will need to classify the current waves transmitted in this way, for testing purposes.

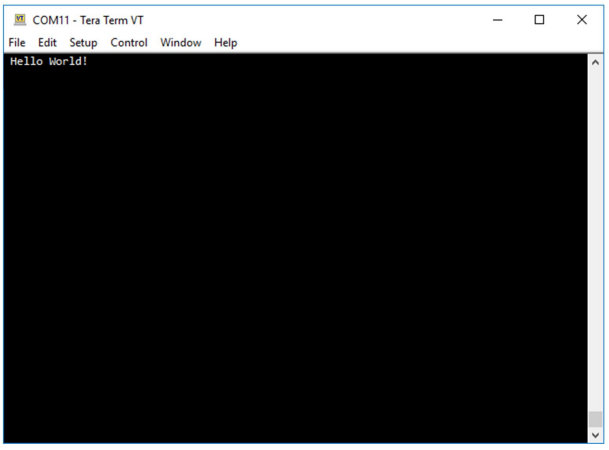


Figure 48: Tera Term Reading Serial Data from USB

3) TIDA → LaunchPad → PC

The final step is to load the tested code in the TIDA-00929 and to modify accordingly the enabled UART ports. The idea is to transmit data from the TIDA to the LaunchPad, and then forward those same data, again via UART, through the USB port to the PC, where we can verify their correctness with Matlab or Tera-Term. To avoid the need of soldering an external Pin on the TIDA board, we use only the TP9, that is the TX port of the UART interface used and which provides a small hole that can be used for fastening a jumper to connect this TX terminal to the RX of the MSP-EXP430FR5994.

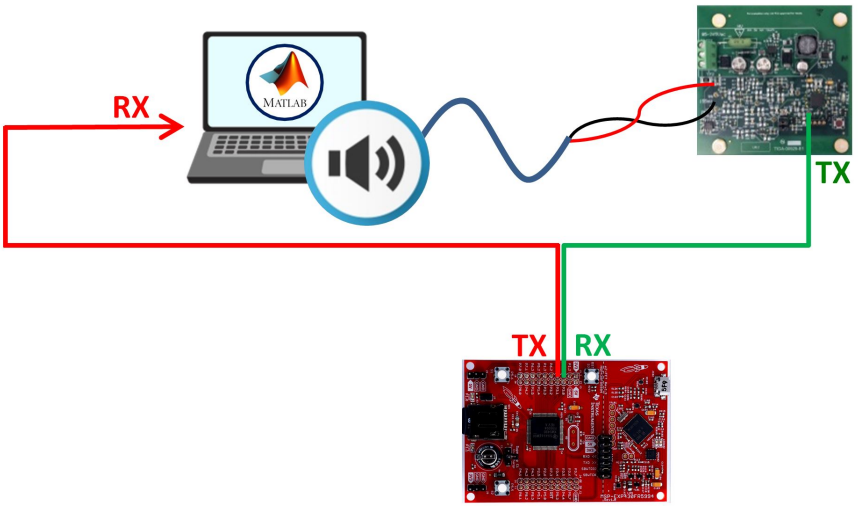


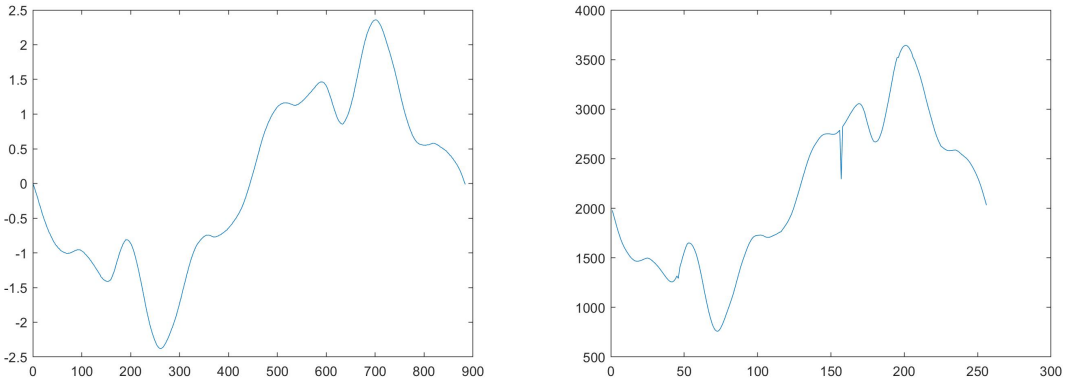
Figure 49: Audio Jack → TIDA → LaunchPad → PC (USB) → Matlab

We proceed by integrating the code of TIDA's UART in its Event Detector one, making the serial transmission happening after an event detection and making, in the first prototype architecture performing the classification in cloud, the acquired and isolated waveform period the data transmitted. In order to be able to transmit the $Q15$ type data, which is used in the Event Detection code to represent the acquired wave and which is made of 16 bits (2 Bytes), we need to separate each sample into two successive 8-Bits transmissions, as shown in the following code snippet .

```
for(k=0;k<256;k++){
    for(i=0;i<2;i++){
          if(i==1){
                TX_Buffer = (uint8_t)(Data_To_Transmit[k] >> 8);
                }
          else if(i==0){
                TX_Buffer = (uint8_t)(Data_To_Transmit[k]& 0x00FF);
                }
          UCA0TXBUF =TX_Buffer;
          __delay_cycles(10000);
    }
}
```

We use Matlab to read the data coming from the USB, transmitted by the TIDA and forwarded by the Launchpad, we are able to directly select a *uint*16 type of data and the two separate bytes will be automatically merged into a 16-Bits integer.

After receiving from the USB the test current wave signal which has gone through the TIDA, LaunchPad and PC UART channels, we collect the wave period and we feed it into our appliance recognition neural network. As we can see in Figure 50, in which we compare the database ideal wave period and the one transmitted via UART, the two waveforms are very similar. Moreover, we need to take into account the fact that, if we will perform the FFT in Cloud with Matlab, we are also able to apply any kind of pre-processing to the received wave.



(a) Database Waveform

(b) UART Received Waveform

Figure 50: Waveforms Comparison

Finally, feeding the Waveforms received via UART to the Neural Network for the appliance recognition, we obtain correct classifications. We can, therefore, conclude that the data transmission via UART between the boards does not alter the signals information and that this communication mechanism is adequate for our application. The UART Transmission analysis conducted so far lays the foundations for the possibility of transmitting the isolated wave period or its spectrum to a Wi-Fi board for the successive Wi-Fi communication, respectively, of that same isolated period or of the final classification result, according to which one of the two prototype architectures we are dealing with.

5.2 UART Transmission Issues

The main issues during the UART transmission chain is encountered in the UART communication between the TIDA and the LaunchPad, which resembles the actual final communication system between the TIDA and the Wi-Fi Board. In fact, frequently, the data transmission appears to be erroneous, for a loss of data. However, thanks to the different debugging steps previously performed, we can assume that the transmission and the reception codes loaded in the two boards are fully working.

Hence the problem is due to the use of non-shielded and fairly long wire for the UART transmission between the boards.

The data that we are transmitting from the TIDA, regarding the samples of the current wave acquired by the ADC, are expressed with 16 bits, while with the UART we transmit only 8 bits at the time. Consequently, every signal sample is delivered with two subsequent UART TX, by sending, separately, firstly the 8 LSBs and then the 8 MSBs.

In the case of the classification in cloud prototype, in order to transmit an entire signal waveform, we need to transmit 256 samples, hence 512 bytes. Although we are able to correctly deliver most of the bytes, if we miss even a single one of them, it will result in a compete erroneous interpretation also of all the following data, as can be visualized in Figure 51. This is due to the fact that, at the receiving stage, to recreate the 16-Bits samples, that we are transmitting as two successive bytes, we multiply the 8-MSBs by 256 (2^8) and then we add the 8-LSBs value.

If we miss a word, we will end up doing this operation with non-coherent values, obtaining results that are extremely wrong, and moreover this problem will continue for the rest of the transmission.

Samples Data Stream	1	2	3	4	5	6
u-int 16 [TX DATA]	1968		2043		3245	
Binary	1011 0000	0000 0111	1111 1011	0000 0111	1010 1101	0000 0111
u-int 8 [TX DATA]	176	7	251	7	173	12
u-int 8 [RX DATA]	176	MISSED	251	7	173	12
u-int 16 [RX DATA]			64432		44295	

Figure 51: 16-Bits Data received with 1 Word Missing

To solve this issue, we need to introduce, at the transmission stage, a certain level of redundancy, in order to detect the right byte value at each transmission and prevent the occurrence of phase displacement of the data stream due to missing data.

In particular, we introduce three levels of redundancy, repeating the UART transmission of each byte for a total of three times. One of the advantages of this technique is the easiness of C code adjustment, since we simply have to insert an additional *for cycle* in the transmission part of it.

u-int 16 [TX DATA]	176			7			251		
u-int 16 [TX DATA] [REDUNDANCY]	176	176	176	7	7	7	251	251	251
u-int 16 [RX DATA]	176	176	983	7	MISSED	7	251	251	251

Figure 52: Redundant TX with Missing and Wrong words

On the other hand, the Matlab code for the reception of the redundant data and its interpretation, providing robustness to either missing or wrong word transmission, is not a trivial task.

5.2.1 Matlab Redundant RX Algorithm

The Matlab Algorithm for the reception of 8-bits data, with a level three of redundancy, and their correct reconstruction in 16-bits data, is based on the analysis of three subsequent received data that we call i, i+1 and i+2.

Starting from these three words we need to verify which one of them are equal, so we check for i=i+1 ; i=i+2 and i+1=i+2. Now we analyze the different possibilities that can occur during the transmission of these three subsequent bytes, that should be redundant.

- The three words are equal → Correct Transmission.
- Missing Byte → In i+2 we have a different word, but i+2 = i+3 or i+2 = i+4.
- Wrong data received → In any position i have a different data ≠ i+3 and i+4

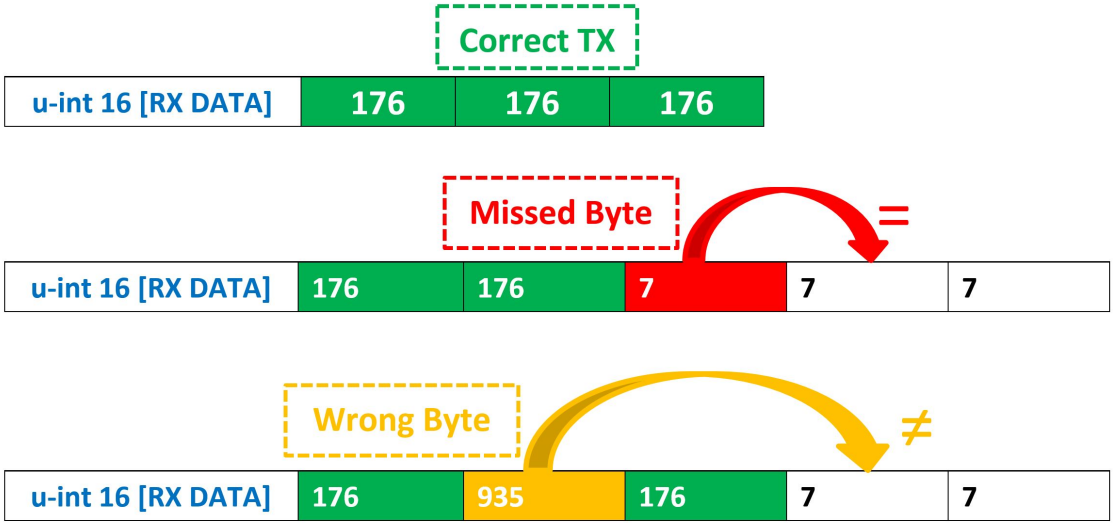


Figure 53: Matlab Algorithm for Redundant RX

Therefore, knowing the equalities between the three analysed bytes, and also looking eventually at the next ones, we can evaluate whether we have a correct transmission; a missing byte or a wrong data reception and eventually retrieve the right data.

5.3 Matlab GUI - Read Signal from COM

In order to simplify the process of data acquisition from the UART, its graphical comparison with the original waveform coming from the Database and the classification of the received signal, we implement exploiting the Matlab App Designer tool a simple GUI (Graphic User Interface).

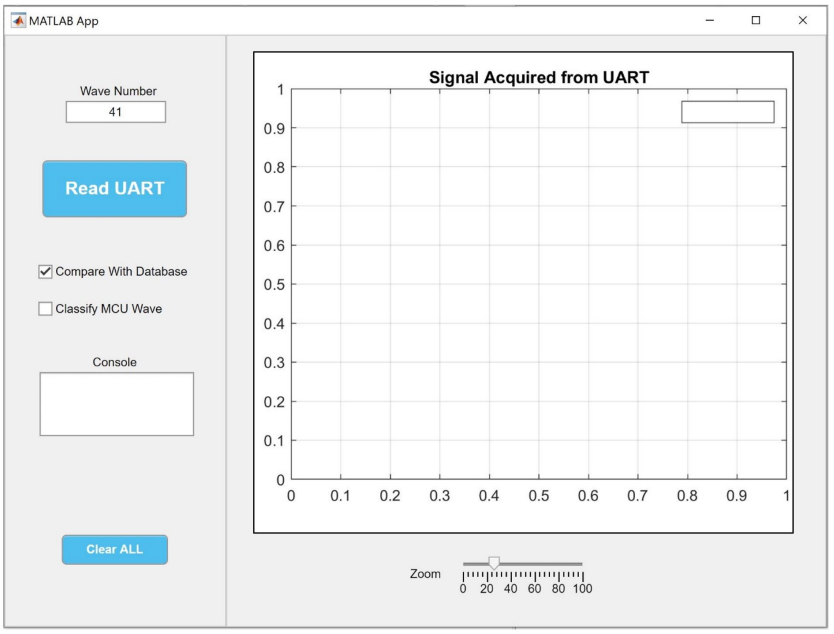


Figure 54: Matlab GUI

From the GUI it is possible to start to read data from UART, by pressing the corresponding button. Before doing that, it's necessary to indicate in the box on top the number of the appliance under test, in order to be able to show the right wave from the database and verify the correctness of an eventual classification. After pressing "Read UAR", the Message "Waiting for TX..." will be displayed in the Text-Box provided below. After a know strating code is detected, we read from UART the 512 Bytes with a three degree redundancy that compose the signal information to be received.

After the completion of the transmission we extract, using the algorithm described in Section 5.2, the valuable information, removing the redundancy and eventual errors. Finally we can merge the 512 Bytes retrieved into the 256 16-Bits words that originally composed the waveform data.

By checking the two boxes present in the GUI ("Compare with Database" and "Classify MCU Wave") we can, respectively, compare graphically the UART-received waveform with the database one and classify the former one. In this case, in the Text-Box, will appear the result of the classification as "CORRECT" or "NON CORRECT".

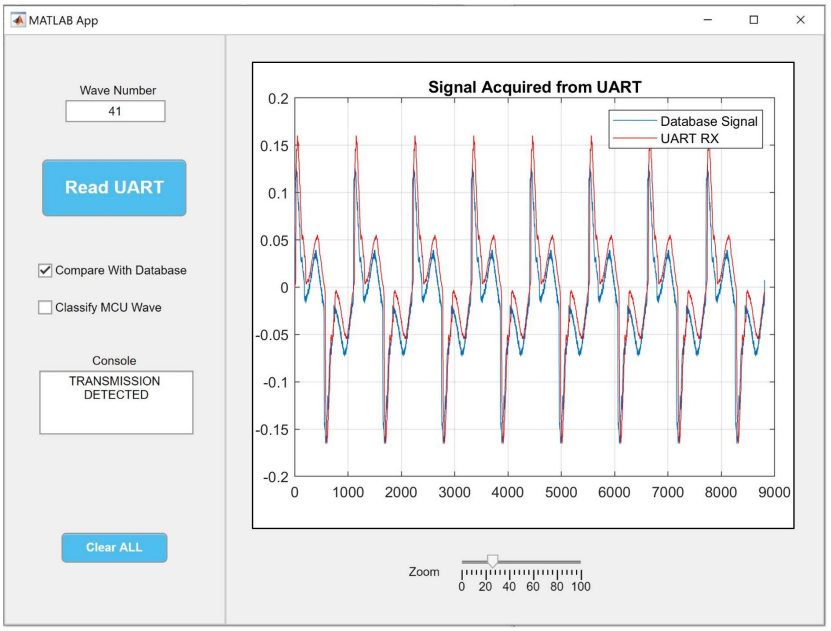


Figure 55: Matlab GUI in action

6 Wi-Fi Communication

Now that we have built the C-Code for a fully functioning UART communication system from the MSP430, we need to explore the possible solutions for implementing a Wireless communication using an additional board still to be defined,

In fact, the main idea, is to acquire the current wave signals using the TIDA board and then, in the case of classification in cloud, to transmit the measured signal via UART to a WiFi Board, which is used to forward those data to the cloud, where we can properly process it and perform the recognition of the relative appliance.

On the contrary, in the case of classification in edge the idea is to perform the FFT with the MSP430 in the TIDA, and then transmit, similarly to the previous case, the spectrum of the signal, to the Wi-Fi board, where we perform the actual classification and from which we can directly transmit the final result to other devices. In this case the Wi-Fi communication is only responsible of the distribution of the outcomes.

Since we are interested in transmitting the acquired data to a remote cloud or directly to other devices, the best solution appears to be the use of Wi-Fi, exploiting a local Access Point nowadays available in any home. In fact, other solutions, like the use of BLE (Bluetooth Low Energy), which can transmit up to 1 Km of distance and which presents a much lower power consumption with respect to WiFi, or others communication protocols like ZigBee, will need the use of an additional HUB to transmit to remote devices and clouds.

Figure 56: Communication Protocols

In order to allow a direct communication between the Wi-Fi Board and the Cloud in one prototype architecture and to allow an easy distribution of the results toward other devices in the other, we decide to use an MQTT protocol, which is based on a remote Broker Server.

6.1 MQTT Protocol

The MQTT (Message Queuing Telemetry Transport) network protocol is based on a publish-subscribe structure, for which each device can create a topic and/or subscribe to it and publish a message relative to that. All the other devices subscribed to that same topic will receive the message. MQTT, is a typical IoT protocol that provides a lightweight protocol, which runs over TCP/IP and does not require a wide bandwidth.

MQTT protocol relies on two types of entities: the Broker and the Clients. The Client can be represented by any device which is connected to the Broker. It is allowed to create topics, to subscribe to them and to send messages regarding any topic. In our example, the Client is represented by the Wi-Fi Board that transmit the previously acquired waveform to the Broker. The Broker is the remote server, in our project it is the Laptop or any other appropriate device, which receives all the different messages, related to specific topics, from all the connected clients and redirects those messages to all the clients that are subscribed to the corresponding topic, notifying them.

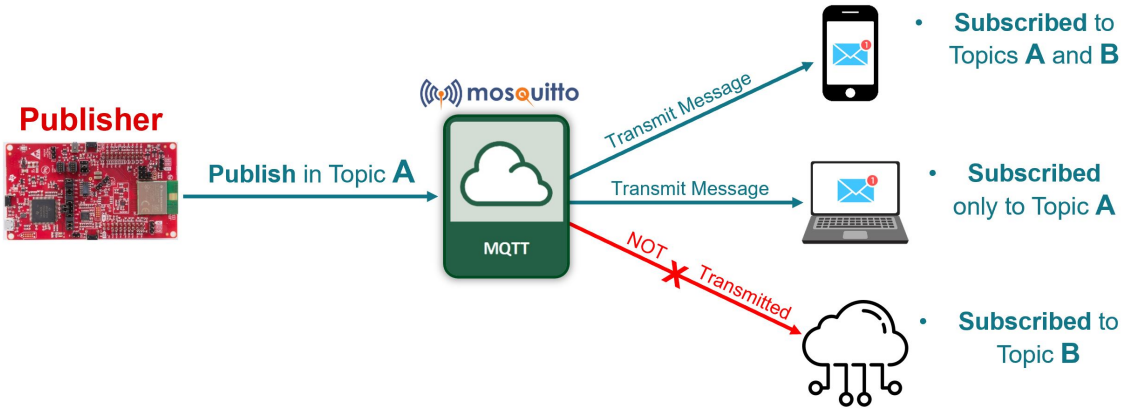


Figure 57: MQTT Structure

The MQTT protocol has been chosen because it allows an easy communication mechanism with a remote server and it is supported by the SDKs (Software Development Kits) provided with a large number of Texas-Instruments Boards that allow Wi-Fi communication. In addition, due to the small amount of data that we need to exchange, MQTT does not provide any relevant limitation in terms of maximum message length.

6.2 Wi-Fi Board Choice

After considering all the necessities for the project, our choice for the Wi-Fi board falls on a CC3220MODASF, which is a LaunchPad, based on a 32-Bits ARM Cortex-M4, made specifically for wireless communications.

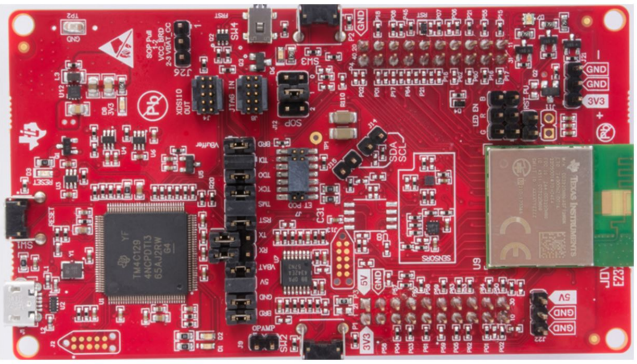


Figure 58: LaunchPad CC3220MODASF

The CC3220 LaunchPad provides a full-duplex UART channel, which is needed in the project for the communication with the MSP430, and a USB port for the power supply and to allow a direct communication with the PC to make the debugging phase much easier.

In addition, for the CC3220 LaunchPad it is available a wide SDK, providing diversified code examples that we can use as foundations of the program.

We can also find a specific code example for the implementation of an MQTT-based communication that we will use as starting point.

6.3 MQTT Transmission Implementation

6.3.1 SDK Code Analysis

In order to run properly the Software Development Kit MQTT code example, we need to modify accordingly the parameters indicated in it, such as the Access-Point Name, the Password and the type of security associated (Open, WPA/WPA2).
The code example allows the board to connect to the specified access point and, by default, automatically subscribes the device to two MQTT topics, called "Broker/To/cc32xx" and "cc32xx/ToggleLED1". Therefore, every time a message will be published in those two topics, the LaunchPad will be notified and will read it.

The board also provides the visualization of its status and, eventually, of the messages published in the topics of subscription, by sending data through the UART channel of the USB port, already connected to the PC. Thus, by setting up Tera Term accordingly with the specified Baud Rate of 115200, we are able to consult the data, as shown in Figure 59.

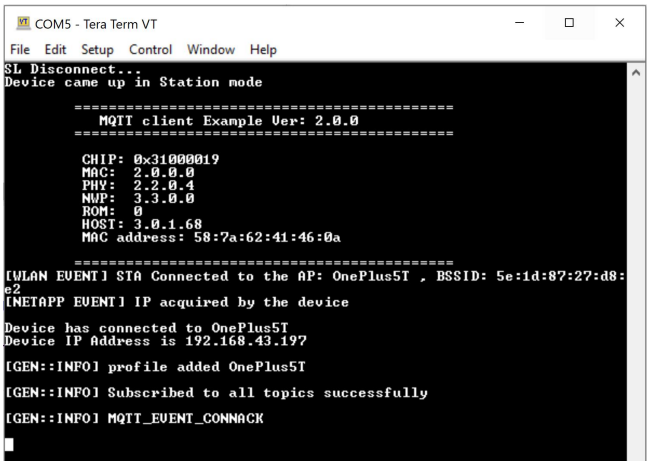


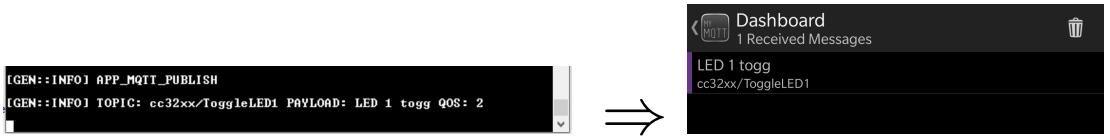
Figure 59: CC3220MODASF MQTT UART Console

By pressing the Switch 2 on the LaunchPad we observe the toggling of the Blue LED on the board and a message is published in the topic cc32xx/ToggleLED1.

Figure 60: Message received on console

Consequently, in the UART console we can visualize respectively: the topic in which a message has been published; the payload, that is the actual message; and the QoS (Quality Of Service), indicating the number of acknowledgments required for the transmission.

To test furtherly the MQTT message transmission, we connect to the same Broker another device -a smartphone- and we subscribe it to those same two topics. After that, we verify that the messages published by the LaunchPad are visible from the phone and vice versa, testing at the same time the functioning of both the exploited topics.



(a) Message Sent by the LaunchPad (b) Message Received by the Phone

Figure 61: Message Transmission LaunchPad → Phone

(a) Message Sent by the Phone (b) Message Received by the LaunchPad

Figure 62: Message Transmission Phone → LaunchPad

6.3.2 Customization of the Code

For our purposes, we need the Wi-Fi board to be capable of receiving data via UART and to publish in a certain topic of the MQTT Server to which it is connected. The information sent to the Broker are thus redirected in case of classification in cloud, to our the remote server, the PC, where they can be processed to perform the actual classification; while in case of classification in edge, the MQTT Broker will simply redirect the recognition outcome directly to other devices.

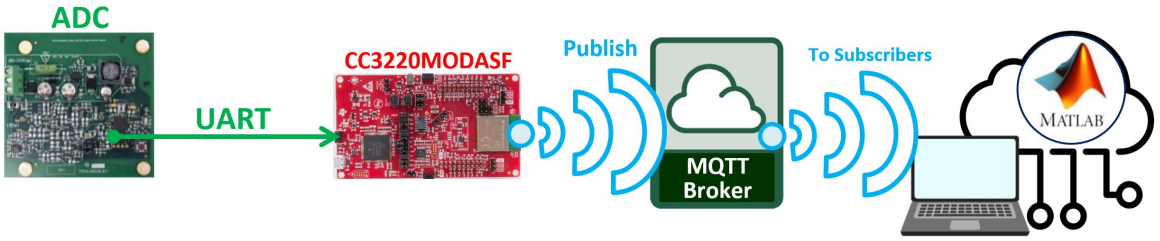


Figure 63: MQTT Publish-Read Chain

At the remote server (PC) level, in the first case, we are able to read the messages that have been published by the Wi-Fi board, in the predefined topic sections, by using Matlab and an apposite MQTT Data Exchanger API.

To do so, we build a script capable of subscribing to the same MQTT topics in which the CC3220 is writing the data received from UART; verifying when there is a new message in them and, in case, reading it. The described Matlab script is reported in the Section 10.2.3.

To verify the functioning of the implemented C and Matlab code and to make their debugging easier, we do not connect the TIDA part of the chain yet, and we simply proceed checking whether from Matlab it is possible to read the messages published by the CC3220 LaunchPad and by an additional test device, a smartphone.

After that, in order to verify the correct functioning of both the UART and the Wi-Fi communication mechanisms, we modify the C code loaded into the CC3220 so that the message that it publishes in the MQTT topic is the one that it receives from the UART channel. At this point, we are introducing the operation of reading from the UART but we are still not using the TIDA board, since the UART channel from which the LaunchPad reads is the one present in the USB already connected to the PC, from which we transmit a test message.

Consequently, we can easily write in this UART channel by typing on the same Tera Term console in which we visualize the status of the LaunchPad and the messages we receive from it.

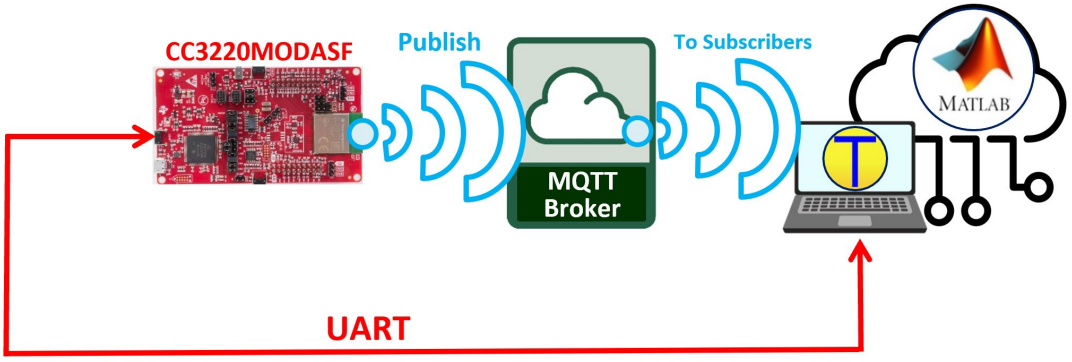
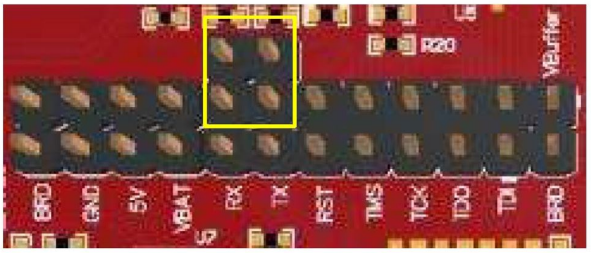
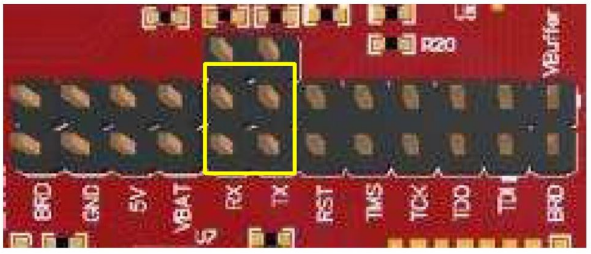


Figure 64: MQTT Testing Chain

By exploiting the external Pins on the CC3220 and the available jumpers, we are able to redirect the UART channels of the board with the USB COM port, to the standard GPIO pins.



UART routed to USB COM Port



UART routed to the 20-Pin Connection

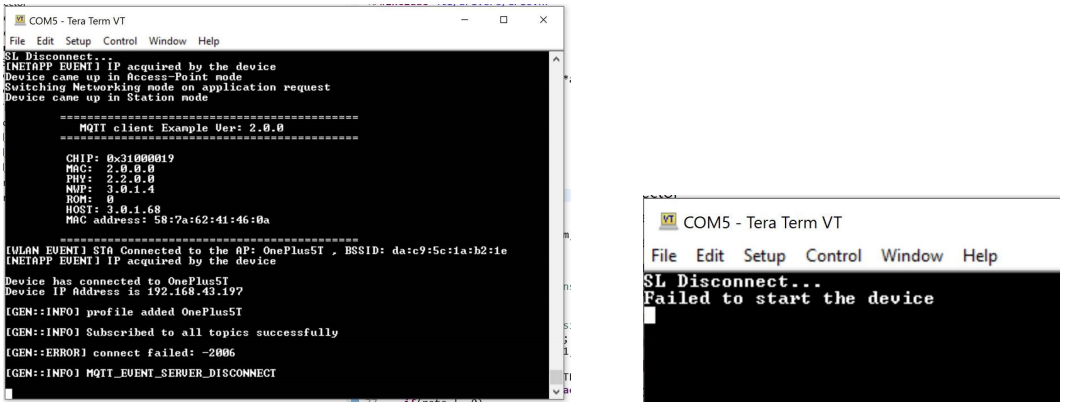
Figure 65: Jumpers configurations

In this way, we are able to connect the RX channel of the UART to the TIDA-00929, still keeping the TX UART channel connected to the USB, so that we can still consult the UART-Console by mean of Tera Term.

6.3.3 MQTT Communication Issues

The main issue encountered when implementing a Wi-Fi communication with the CC3220MODASF, regards the capability of the board of connecting with a Broker Server.

Initially, we relied on a standard and public MQTT Broker, such as the one provided by eclipse (*mqtt.eclipse.org*), but if in our C code we provided the server address as its URL, the board seemed to be unable to establish a connection or, at least, it disconnected immediately.



(a) Board Immediately Disconnecting (b) Board Not connecting

Figure 66: Errors in Connection with Broker

This issue is solved by performing a punctual "*Sniffing*" of each TCP packet exchanged between the CC3220MODASF and the Access Point at which it was connecting. This is done by mean of the software Wireshark.

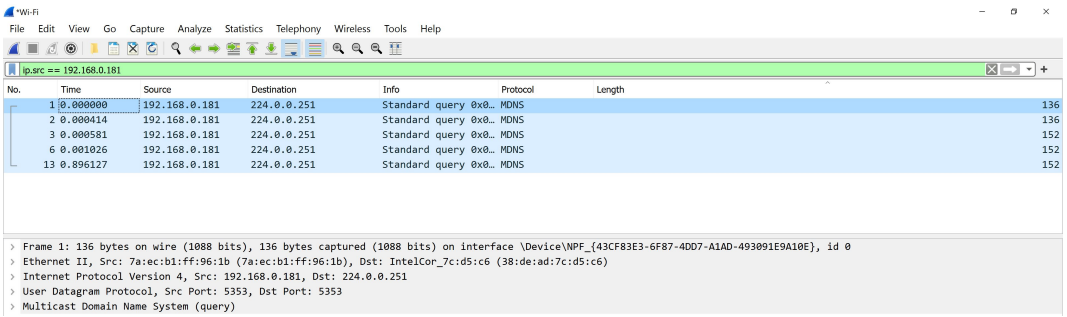


Figure 67: Wireshark

By comparing the IP Address at which the requests of the Board, through the

Access Point, are directed with the actual IP Address of the Broker Server, we can point out that the board was not targeting the right Address.
Consequently, we modify the C code by stating explicitly the IP4 Address of the target broker server.

6.3.4 Private Secure MQTT Broker

Now that we are able to connect out CC3220MODASF, through an IP4 Address, to a public MQTT Broker Server, our purpose is to implement our own private and secure Broker to rely on.

We exploit the Eclipse Software *Mosquitto*, which provides a *Windows* Service to enable the PC as a Broker Server.

Figure 68: MQTT Custom Broker Software

With Mosquitto we are, indeed, also able to setup a secure connection between clients and the Broker by mean of a Password and of an arbitrary list of accepted Client-IDs, to control the access to the Server. It is worth mentioning that, in order to use our device as a Broker Server, we need to allow external connection through the port specified for the MQTT data transmission.

To verify the functioning of the Broker Server by itself we exploit another software which is *MQTT Explorer*, with which can connect with other devices to the Mosquitto server, publish and receive messages.

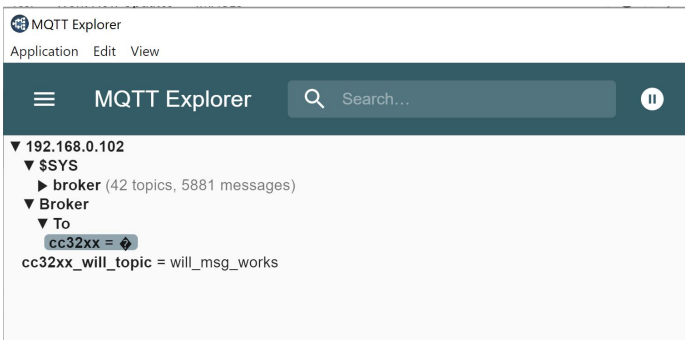


Figure 69: MQTT Explorer

6.4 MQTT Connection Implementation in Matlab

In order to connect with the MQTT Broker and implement a Wi-Fi data transmission between the CC3220 and the PC, we exploit the Toolbox "MQTT Data Exchanger" that can be added to Matlab and which is based on an MQTT API.

The Toolbox provides a simplified set of commands to connect to a Broker, to subscribe to topics and to publish/receive messages. In fact, by using the *mqtt*() command, specifying the IP4 address of the Broker and the communication protocol, in our case TCP, it is established a connection with the MQTT Broker and we can subscribe to a topic by using the apposite *subscribe* command. This function returns a struct-type in which are indicated, among other things, the number of messages currently present in the specified topic that we have subscribed to.

By constantly monitoring this number with a polling mechanism, we can tell when a new message has been published in the topics of interest and we can read it in the format of a string, since the message that would be published by the Wi-Fi board, representing the waveform acquired by the TIDA-00929 and transmitted to it via UART, is made of *Unsigned Char* values (1 Byte size). The string is then converted into an array of *uint*8 and, finally, from the 8-Bits data recevied we can retrieve the 16-Bits that represent the actual waveform acquisition values.

This can be achieved by using, for two consecutive *uint*8 received values that compose a single *uint*16 data, the following expression, where the i term represents the LSBs of the 16-Bits value and the i+1 term its MSBs, shifting the latter one to the left by 8 Bits and then adding the former one to the result:

$$data\_uint16 = data\_uint8_i + (data\_uint8_{i+1}) << 8$$

The full Matlab script for the MQTT connection, message extraction and 16-Bits data construction is reported in the Appendix 10.2.3.

6.5 Matlab MQTT API Issues

It is worth mentioning that, due to the particular data types used in the MQTT Matlab API, we encountered an issue regarding the interpretation of Matlab of some particular values when retrieved from the MQTT Broker messages.

In particular, we noticed that, when we performed a Wi-Fi transmission through the MQTT protocol, the message read by Matlab presented some errors. At first, we hypothesized that the errors were caused by an erroneous UART transmission between the TIDA-00929 and the CC3220MODASF or by a wrong code implementation for the publishing of the waveform values from the CC3220 to the MQTT Broker.

However, performing a cross-analysis, using both Matlab and Code Composer Studio, of the data transmitted with redundancy by the TIDA then received by the CC3220 and sent to the MQTT Broker and finally received by Matlab, we noticed that, when there was the occurrence of a wrong value, this was repeated in the same way also in all its redundancy repetitions.

This led to an exclusion of the possibility of an erroneous UART transmission. In addition, we analysed punctually the Bytes sent by the CC3220 to the MQTT Broker, by also exploiting Tera Term and making the CC3220 print in it the same data transmitted to the Broker. Thus, we could also exclude the possibility of an erroneous transmission CC3220 $\rightarrow$ MQTT Broker.

As a consequence we isolated the issue as part of the MQTT Broker $\rightarrow$ Matlab communication section.

Therefore, we proceeded by publishing in the MQTT Broker known sequence of all the possible 8-bits value, from 0 to 255, verifying then the values retrieved by Matlab. We noticed that, for some particular values read from the received MQTT messages, Matlab was interpreting them in an erroneous way.

In particular, the transmitted values that are systematically misinterpreted are the ones between 128 and 159. However, we also observed that these values, even though they are read in the wrong way, they are always misinterpreted in the same way, which means that we are able, in Matlab, to manually correct them, since there's a direct correspondence between the wrong values read and the actual ones. The conversion table adopted is reported in Table 3.

Right Value	**Misinterpreted Value**
128	8364
130	8218
131	402
132	8222
133	8230
134	8224
135	8225
136	710
137	8240
138	352
139	8249
140	338
142	381
145	8216
146	8217
147	8220
148	8221
149	8226
150	8211
151	8212
152	732
153	8482
154	353
155	8250
156	339
158	382
159	376

Table 3: Conversion Table

For those values that are represented by the same "misinterpreted" one, we computed them as the average of the previous and the following data.

Subsequently, we build a Matlab script to automatize the audio stream of the

current waves from the PC to the ADC board and the acquisition of the waveforms, isolated by the TIDA, transmitted via UART to the CC3220MODASF and sent via WiFi to the Broker, from the MQTT server.

In particular, this Matlab script, reported in Appendix 10.2.4, creates the 1050 audio files from the same number of appliances measurements and then it plays them through the audio jack one at the time. After that it waits for new messages, containing the Waveform transmitted by the CC3220, to be published in the MQTT Broker topics, and re-builds the received signal. At this point the script plots the profile of the acquired signal and the user can indicate whether the waveform has been correctly received or we need to repeat the operation.

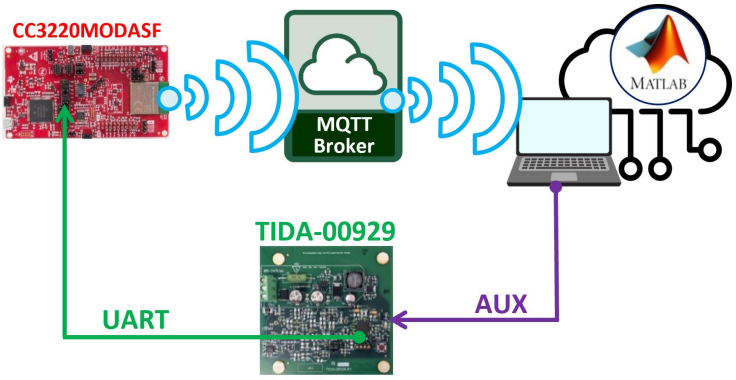
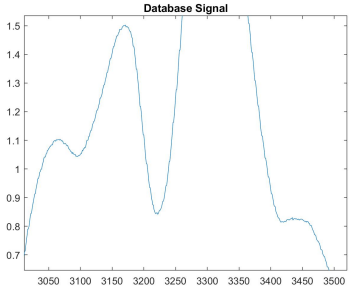
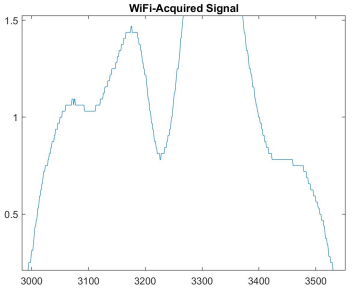


Figure 70: Wi-Fi Testing Configuration

We are able to collect in a fairly short time an entire data set of the appliances considered which reflects much better the actual measurements that we will need to classify in operative conditions rather than using the plain waveforms contained in the database. In the data collected in this way, indeed, we have the real ENOB and distortions, phase noise and errors that will be encountered in the actual classification cases. In this way we will be able to train our neural network accordingly, to significantly increase the prediction accuracy.



(a) Ideal Database Signal



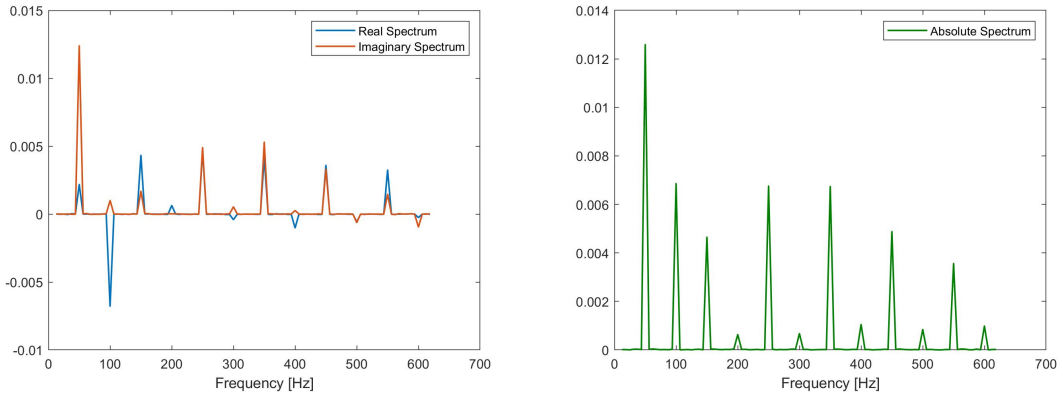
(b) Signal acquired from WiFi

Figure 71: Errors in Connection with Broker

7 Features Analysis

7.1 Absolute Spectrum Classification

Now that we have ensured that it is possible to communicate via WiFi, through an MQTT Broker, the next step is to try to verify whether it is possible to exploit, to perform the classification, the absolute value of the wave spectrum instead of the separated Real and Imaginary ones.



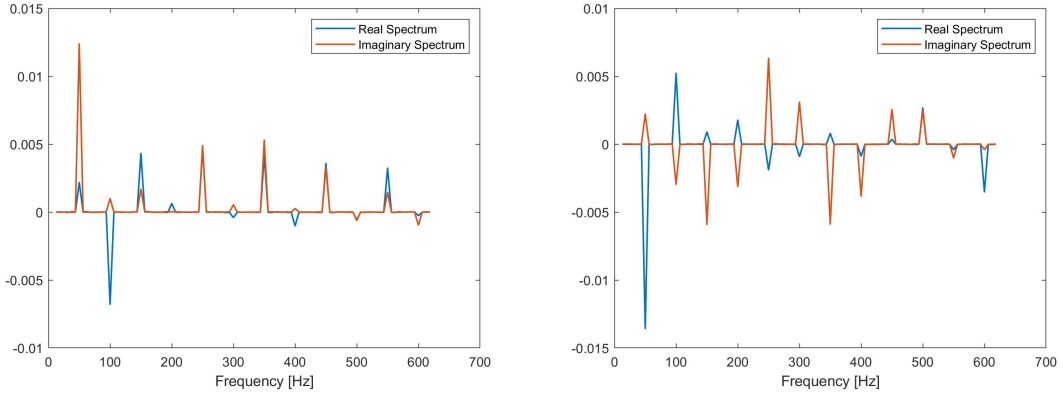
(a) Real and Imaginary Spectrum

(b) Absolute Spectrum

Figure 72: Spectra Comparisons

Although the use of the absolute spectrum means to have an halving of the number of information to perform the classification, it also leads to several advantages. In particular, it would make the classification process much more robust to possible errors in the period isolation mechanism that can result in an unwanted and unpredictable phase between different measurements.

In fact, if we use the separate real and imaginary spectra, they are strictly dependent from the phase of the analysed signal and hence of the isolated period. In order to obtain a good coherence in our classification we must provide a known, constant, phase in each isolated period.



(a) Phase 1 (b) Phase 2

Figure 73: Separate Spectra with different Phases

If, instead, we use the absolute spectrum, our measurements and our classification become independent from the phase of the signal acquired and, therefore, our system becomes robust to phase errors in the signal acquisition and isolation.

Moreover, using the absolute value of the signal spectrum will also be extremely useful when we will analyse, in the following steps of the study, the possibility of recognising multiple appliances being active simultaneously. In fact, due to the different typologies of loads that can be connected to the power grid, we will observe the presence of a phase difference between the current waves absorbed by two different appliances, even though the voltage supplied to them is the same in amplitude and phase.

As a consequence, when we have more than one appliance absorbing current from the grid at the same time, the resulting current wave will be made of the sum of the different appliances' ones. Therefore, when we isolate a section of the composed current wave, we will have an unknown and unpredictable phase between the signals that compose the acquired waveform.

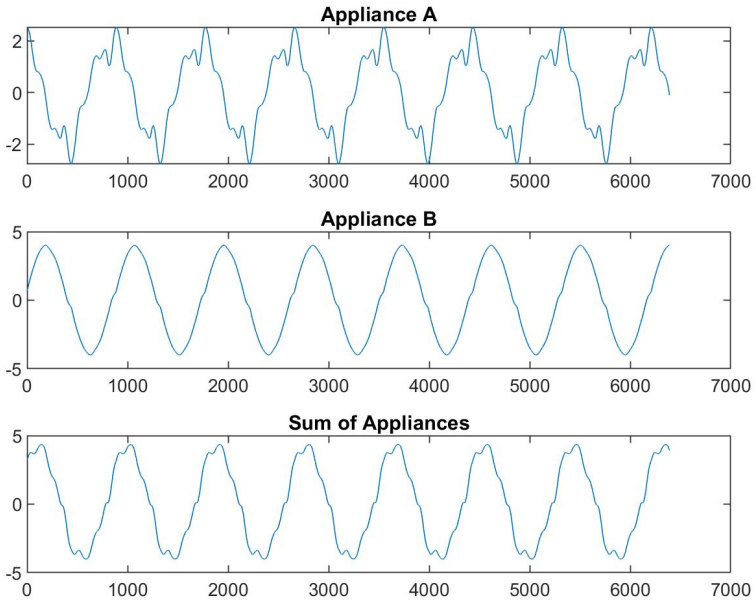


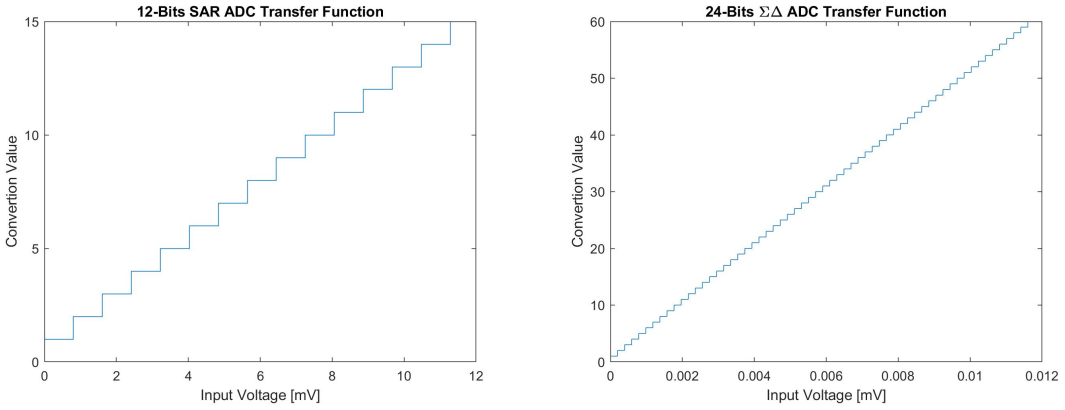
Figure 74: Unknown Phase between appliances

Consequently, a disaggregation of the involved appliances would be impossible using the separate real and imaginary spectra, since the presence of an unknown phase between the two signals will make the spectra measurements non coherent and would lead to an impossibility of its classification.

The analysis of the signals disaggregation will be discussed in Section 8.

7.2 ADC Requirements Analysis

At this point of the study, we need to verify whether in our system it is actually required an external ADC with a number of bits higher than the SAR-based one integrated in the MSP430FR5994. In fact, the latter one provides a 12-bits conversion, with an ENOB of 10-bits, while the one evaluated to substitute it with provides a 24-bits conversion with an ENOB of 18-bits and it is based on a Sigma-Delta Converter.



(a) SAR Transfer Function (b) ΣΔ Transfer Function

Figure 75: Precision Difference between 12 and 24 bits ADC

The advantages of using an higher bit number ADC is the increase in the accuracy and the strong reduction in the quantization noise and the consequent improvement of the SNR_q (Signal over Quantization Noise), since:

$$SNR_q \cong 6.02N + 1.72 \qquad with\ N = Number\ of\ Bits$$

Moreover, quantization noise strongly depends on the amplitude of the considered signal and in particular on whether it exploits the entire ADC dynamic or part of it. In fact, if the signal observed has an amplitude which is lower than the maximum dynamic available, the SNR_q will decrease with it of -20dB/dec, and therefore there's a linear dependency between them.

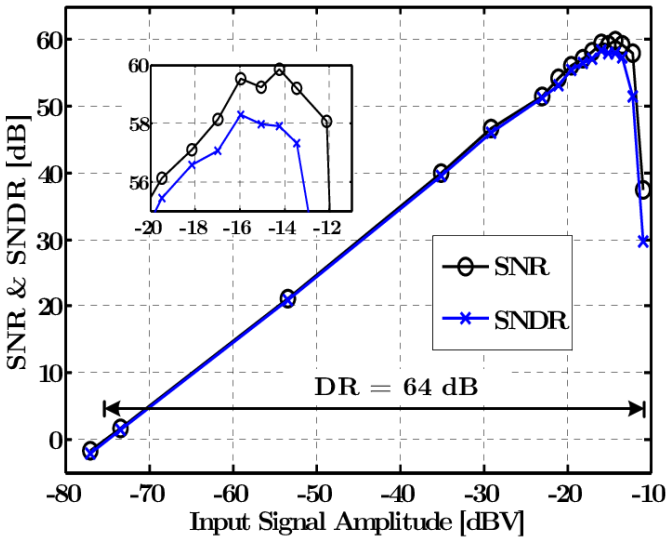


Figure 76: Variation of SNR_q with non full-dynamic signals

This aspect is particularly important for the case of study, since the current waves of the analysed appliances have a wide and diversified range of amplitudes.

However, the use of an higher precision ADC like the one taken into account, would increase remarkably the cost (BOM) of the final device, and hence it is worth analysing whether this modification to the project is actually needed.

To perform this investigation, having in a previous stage of the study verified, with a fixed-point analysis Matlab, that the classification accuracy of the Neural Network remains acceptable for 10-bits data, we build an automatized Matlab algorithm, capable of collecting, directly from the WiFi, the waveforms acquired and converted by the prototype, without any need of external supervision.

The algorithm was built by modifying the one previously developed and reported in Appendix 10.2.4, inserting a wait of 5 seconds between each acquisition and a watchdog to repeat the transmission of the signal at the occurrence of a timeout in case of an occasional miss of the event detection algorithm on the device.

It is important to mention that, in order to make this process more robust against event detection misses and against wrong triggering of it, we modified the FIR filter involved in the event detection by using a much steeper and hence higher order (Order 70), with a Band-Pass behavior between $30Hz$ and $300Hz$, as shown in Figure 77. The filter coefficients were generated using the DSP-Library provided with the MSP430FR5994 Software Development Kit.

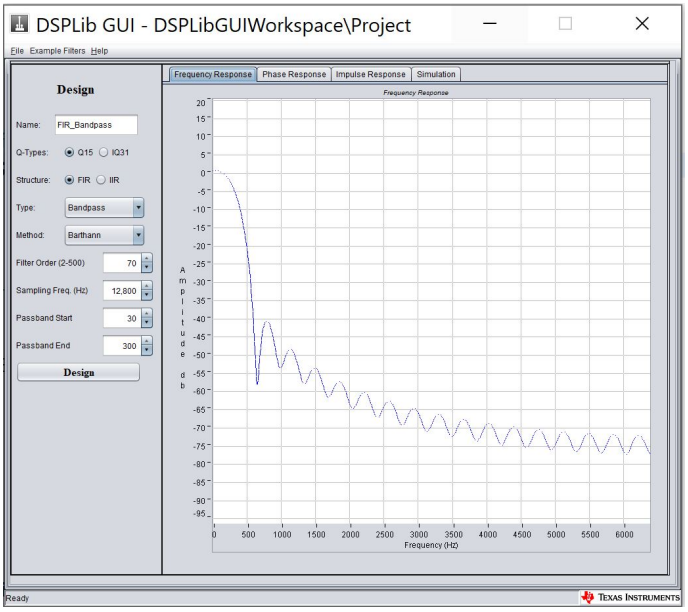


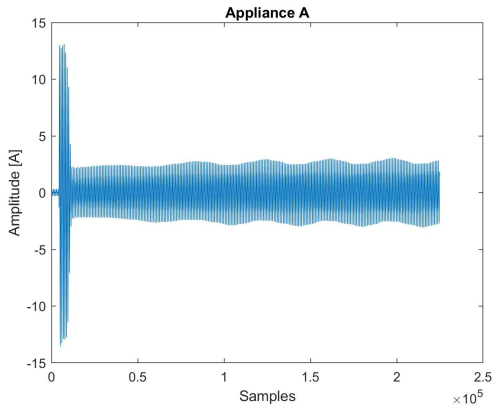
Figure 77: FIR Filter Transfer Function and Characteristics

Therefore, exploiting this algorithm, we are able to re-collect from the WiFi the entire database of 1050 measurements, for a total of 105 appliances. We can now use those acquisition, which are representative of the actual signals in operative conditions, to create some new training and testing data-sets and verify whether the SAR-based 12 bits ADC exploited at this point is adequate. It is worth mentioning that, to make the measurements coherent, the transmission of the waves via audio jack was performed at fixed and constant volume level such that no clipping would occur also for the signals with higher amplitudes.

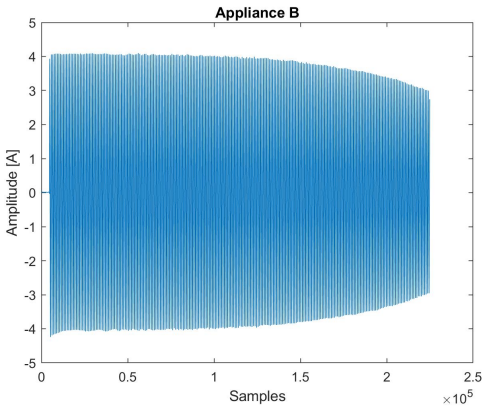
With the new data-set, taken in operative conditions, we obtain an overall accuracy of $\sim$92%, definitely proving that the on-board 12 bits SAR ADC, integrated in the MSP430FR5994, is sufficient for the case of study of appliance recognition.

8 Disaggregation of Multiple Appliances

Now that we have implemented the single appliance classification mechanism, we want to attempt to find an algorithm and a solution to make it possible to recognize additional appliances that are turned on when there are already one or more devices connected to the power line.



(a) First Appliance



(b) Second Appliance

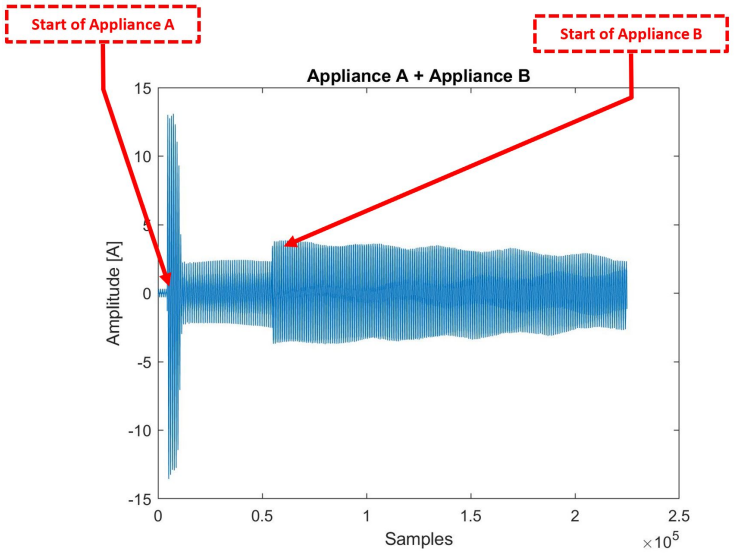
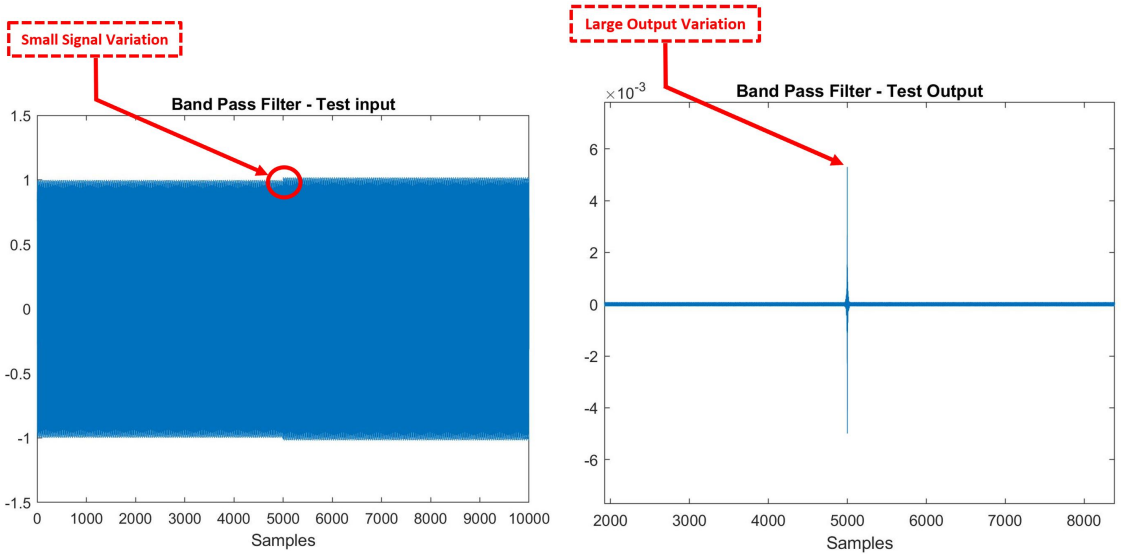


Figure 78: Multiple Active Appliances

For what concerns the detection of the turning on of successive appliances when there are already some active ones, the Band Pass FIR filter, implemented in the MSP430FR5994 and described in the Section 7.2, which is part of the event detection algorithm, is capable of detecting also small variations of its input signal thanks to the accurately chosen behaviour of its transfer function.



(a) Filter Input

(b) Filter Output

Figure 79: Band-Pass FIR filter Behaviour

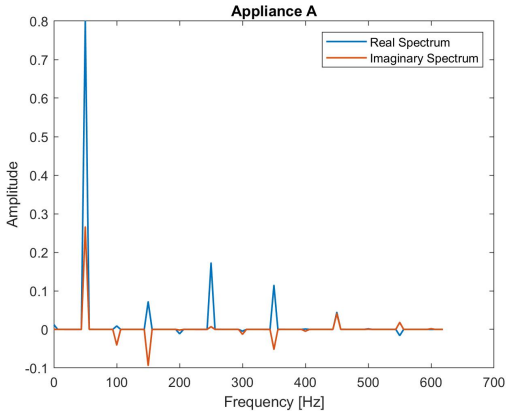
In fact, as we can see in Figure 79 with some test signals, the occurrence of even a small event cause the presence of a clear peak at the output, making possible without any further modification of the firmware built, the detection of additional events while there are already one or, potentially, multiple other devices connected.

8.1 Signals Disaggregation Issues

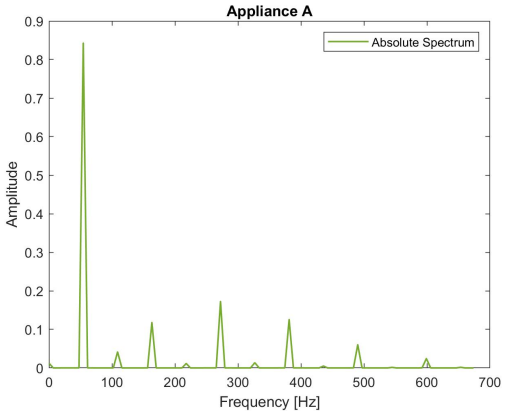
The disaggregation of multiple overlapped signal is not a trivial task. In fact, since each appliance can potentially have a completely different reactive load, it can introduce on its absorbed current wave an unknown and unpredictable phase with respect to the one of the appliances already active on the line.

This leads to the fact that the real and the imaginary spectra of the two overlapped appliances will sum up with an unknown phase between them and therefore the amplitude of their real and imaginary harmonics will be unpredictable as well as the corresponding sum between the appliances. We can't even resort to the use of the absolute spectrum since with the presence of this unknown phase, the spectrum of the sum signal will not be equivalent to the sum of the single appliances spectra, because given two complex numbers X and Y:

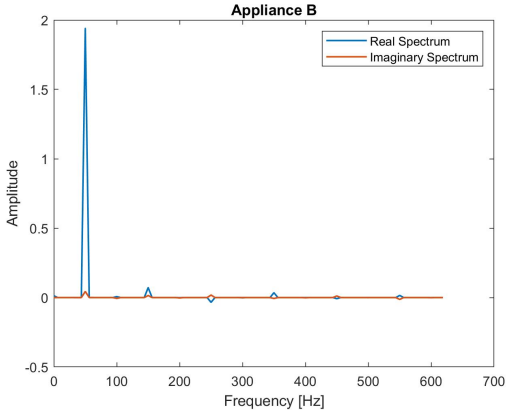
$$|X| + |Y| \neq |X + Y|$$



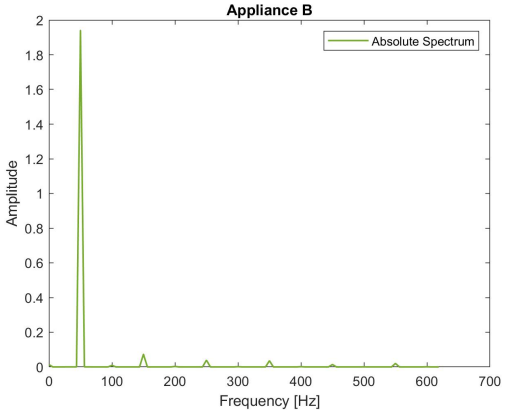
(a) Appliance A Spectrum



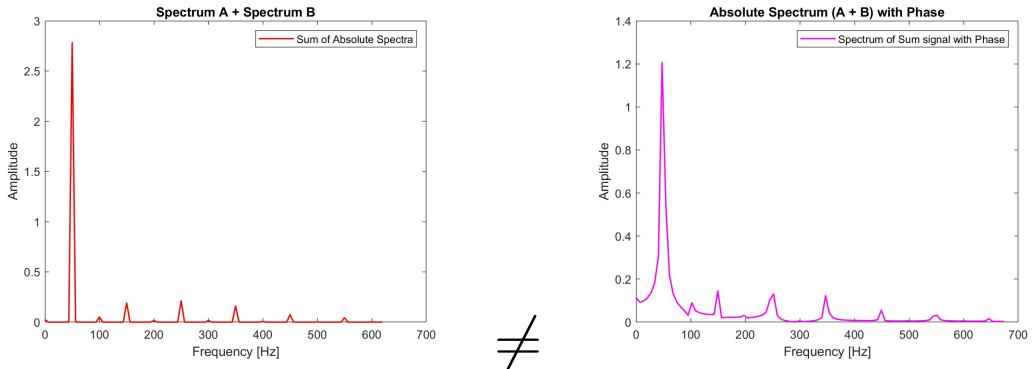
(b) Appliance A Absolute Spectrum



(c) Appliance B Spectrum



(d) Appliance b Absolute Spectrum



(a) Sum of Spectra (b) Spectrum of Sum

Figure 80: Sum of Spectra and Spectrum of Sum comparison

As we can easily notice from Figure 98, the Sum of the two absolute spectra does not coincides with the absolute spectra of the sum of the two signals when there is a phase between them.

As a consequence, when we isolate a period of the overlapped signal, we need to know at least the phase of the already active appliance current wave, to try to remove it from the sum signal. To do that, we can keep track of the number of samples taken between the activation of the second appliance and the isolation of the period of the sum signal and according to the known main harmonic frequency, we can compute the phase of the first appliance with respect to the isolated sum period or, alternatively, we can isolate a period of the sum signal such that it is in phase with the first appliance.

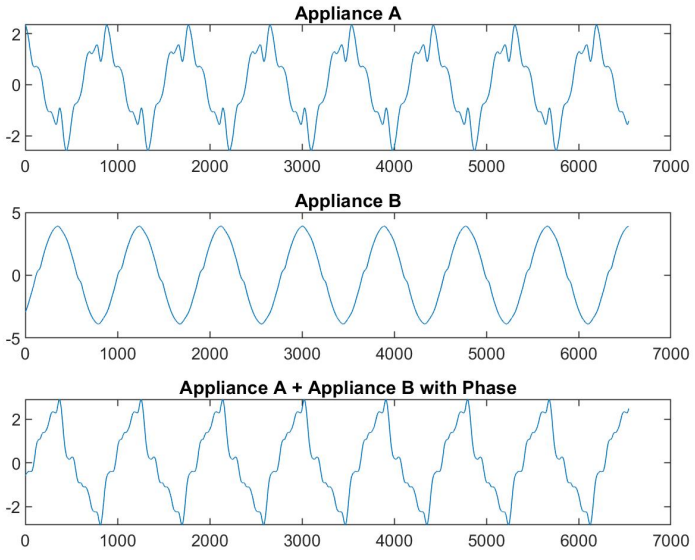


Figure 81: Sum signal with phase

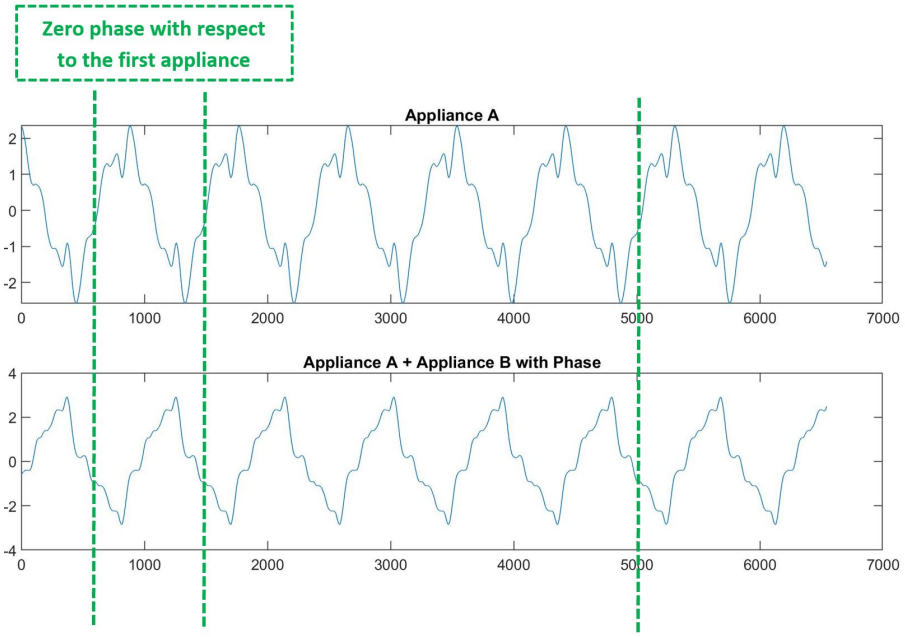


Figure 82: Sum Signal in Phase with Appliance A

However, as we can notice from Figure 82, the current waves absorbed by an appliance can strongly variate their behaviour. This can be due either to a non ideal behaviours of the connected load or of the grid itself, or it can be generated by the fact that many devices, that are generally grouped in the class of the FSM-Based appliances, modify completely their behaviour, their functioning and therefore the trend of the current absorbed in time.

As a consequence, even though we are able to isolate a period from the sum-signal which is in phase with the first e turned on appliance, the spectrum of the first appliance will potentially have undergone a large modification of its behaviour and hence of its current wave. Therefore, we can't simply remove from the in-phase spectrum of the sum signal the spectrum of the first appliance taken at the occurrence of its event to extract the spectrum of the newly activated one.

The best chance that we have to be able to isolate the in-phase sum signal period while knowing the exact trend of the first appliance involved in it, is to take another period from the first appliance just before the occurrence of the second event. Therefore, we can transmit, with the same mechanism described in the previous steps, the newly isolated period from the first appliance right before sending also the one isolated from the sum signal.

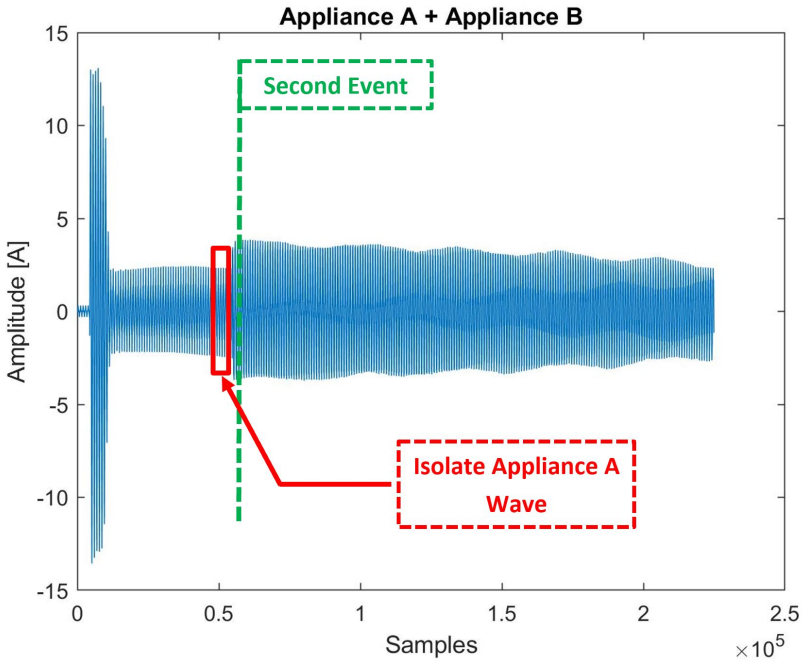


Figure 83: Sum signal with phase

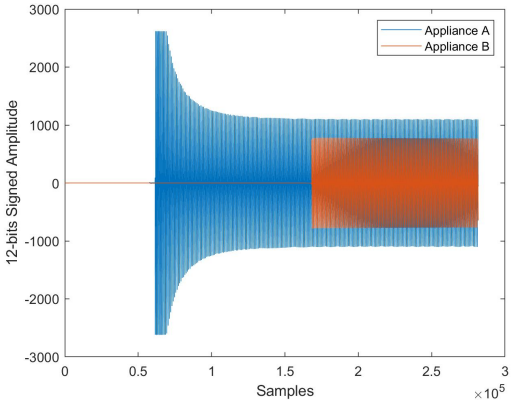
By doing this we significantly reduce the time interval between the acquisitions of the sum wave and of the first appliance waveform that we want to remove from it. Therefore, it will be much more probable that the first appliance current wave will remain unvaried in this time interval, making it possible to extract the second wave and the corresponding appliance classification.

From the Matlab analysis of an extended number of combination of appliances, we find out that the quality of the second appliance extraction and of the consequent results depends on the type of the appliances involved. In particular, we can recognise a total of three possible situations of aggregation of appliances:

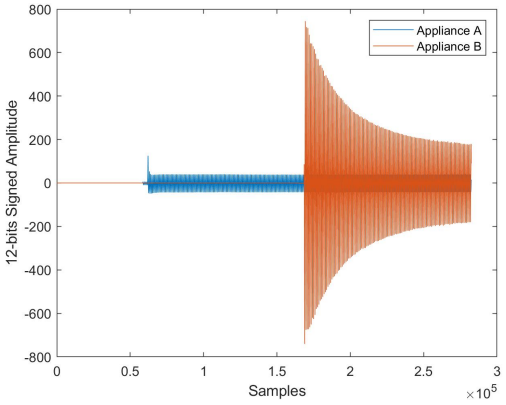
a. Two appliances with comparable current wave amplitudes.

b. Second appliance's current wave much greater than the first one.

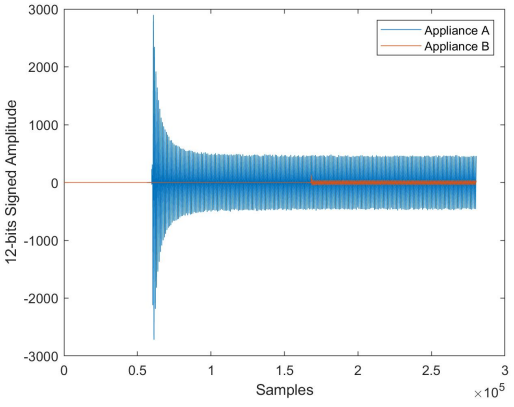
c. First appliance's current wave much greater than the second one.



(a) Comparable Amplitudes



(b) Second appliance greater than the first one



(c) First appliance greater than the second one

Figure 84: Overlapping appliances cases

While the new detection algorithm for occurrence of events is robust to the introduction of multiple appliances and it is capable of detecting with high reliability also the activation of multiple appliances regardless of the relative magnitudes of the current waves involved, the appliance classification one results to be dependent by the situation, among the three described, in which we are operating.

In fact, we observe that, while in the cases of the two appliances having comparable amplitudes or in the case of the second appliance being larger than the first one, which are pictured in Figure 84 - (a) and (b), the disaggregation algorithm is capable of extract effectively the newly activated appliance and we are able to classify it accordingly, in the case of a second appliance with a current wave much smaller than the first one this operation becomes much harder.

If the profile of the first appliance has a small fluctuation between its last acquisition right before the second appliance event and when the sum-wave period is isolated, when we try to remove from the latter one the spectrum of the former, we are not actually removing the new content of the first appliance in the aggregate signal.

Therefore, being the second appliance waveform much smaller than the first one, its error of estimation of the first appliance due to its variation over time may cause a complete misinterpretation of the second one, making it impossible to be classified. The improvement of the algorithm to allow a more robust, reliable and correct classification also in this worst case is left for further studies.

All the considerations and the analysis performed on the case of the aggregation of two devices can be easily extended to the case of multiple appliances, since each one of them is classified individually at its activation and, for the disaggregation algorithm described, we can simply consider the first appliance as the sum-wave of all the devices already turned on, and the second appliance as the newly activated one.

9 Prototypes Realization

The next step of the study is the actual realization of the two prototypes architectures, one performing the classification in cloud and the other one in edge, at local hardware level.

9.1 Appliance Classification in Cloud

The next step is to analyse the realization of the first prototype, the one that performs the classification in cloud, and its overall functioning.
First of all, we need to define whether in this architecture we want to transmit to the Cloud, for the appliance recognition, an entire isolated signal section from the steady state of the original waveform, or if we want to send directly its spectrum. We start analysing the latter hypothesis.

9.1.1 Local FFT for Classification in Cloud

We try to explore the hypothesis of performing locally the FFT on the isolated section of the original waveform and then transmitting via Wi-Fi to the Cloud its harmonic content for the classification. By doing that we would only have to transmit the absolute values of the harmonics of interest, made in total of 8 words of 2 bytes each.

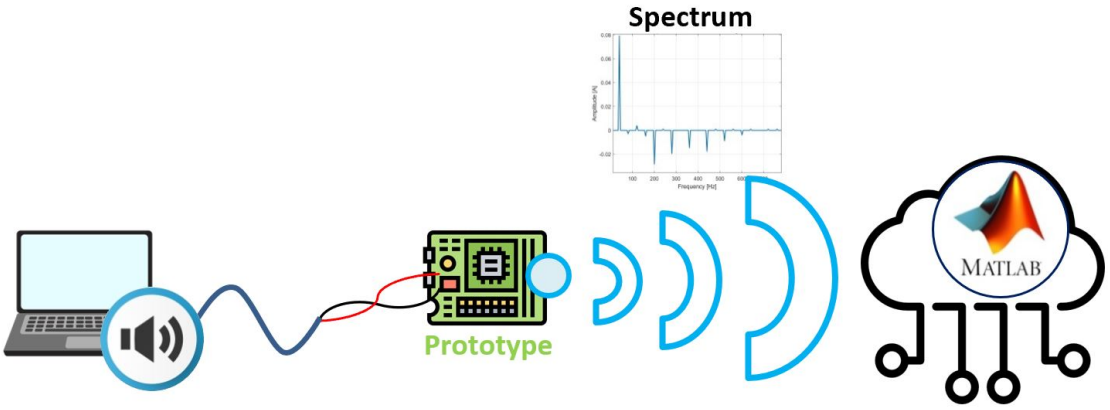


Figure 85: Local FFT configuration

Therefore, starting from the modified Event Detection Algorithm, we want to perform the Fast Fourier Transform on the data saved in the variable that, in the previous discussions, we called *Wave Memory*, in which we store the output of the MSP430 ADC after the detection of an event.

However, even though in the Wave Memory we are able to isolate two complete wave periods, when we perform the FFT on it we encounter the following issues that prevents us from obtaining a reliable and coherent spectrum content:

- The waves extracted always starts at a different point.
- Two periods are not enough to highlight the harmonics present in the signal.

The overall result of the spectrum results to be extremely confused and poor in actual information, as can be seen in Figure 86.

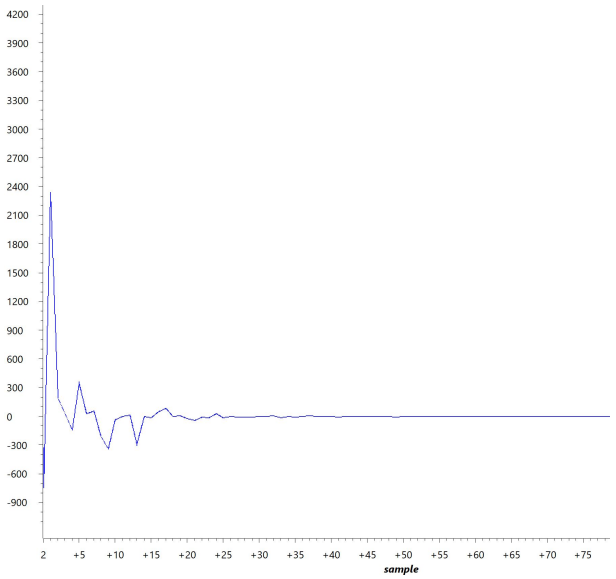


Figure 86: FFT on Two-Periods with Random Starting Point

Once the two main issues that affect the quality of out FFT have been identified, we can try to solve them respectively by:

- Using a Zero-Crossing Algorithm to make the periods start from the same point (e.g. where they cross the Zero) or, as discussed earlier, exploiting the absolute spectrum as the classification feature.
- Increase the length of the waveform section on which we perform the FFT to improve its frequency resolution.

The former, in case of employing of a zero-crossing algorithm, is realized by computing the average of the signal and then recursively checking whether each sample of the waveform is closer to it than the previous one, in order to highlight the array-index of the value closest to the signal average, as shown in the following code snippet:

```
average=0;
for(k=0; k<FIR_LENGTH; k++){
    average += result[k]/256;
}

closest_value=1000;
for(k=0; k<FIR_LENGTH; k++){
    if((average - result[k]< closest_value) && (average - result[k] >0)){
    ZERO_C=k;
    closest_value=average - result[k];
    }
}
```

For what concerns adding more signal periods to the signal for the FFT analysis, we try to extend it to a total of 8 periods, similarly to what we did in Matlab. However, here at MCU level, we face the limited hardware resources that the Micro-Controller presents. In particular, the MSP430FR5994 controller, provides a total of 8 kB of RAM memory, which is also divided in half between the actual processor memory and the LEA RAM, for the acceleration of signal processing operations.

Exploring the functioning of the MSP430, we find out that, in order to perform complex signal-processing operations like the FFT, the involved variables need to be stored in the LEA RAM. Consequently, we need to save the variable, that we call $fftBuffer$ in which we store the signal on which we will perform the FFT and where the FFT function gives its result, in the LEA RAM (4 KB). Since we are sampling at 12800 KHz signals with a 50Hz fundamental frequency, a single period, as discussed in Section 4.4 , will occupy 256 Words, made of 2 Bytes each, for a total of 512 Bytes.

Therefore, the maximum number of signal periods that we can store on our LEA RAM is:

$$Max\ Number\ of\ Periods\ in\ LEA\ RAM = \frac{4096 Bytes}{512 Bytes} = 8\ Periods$$

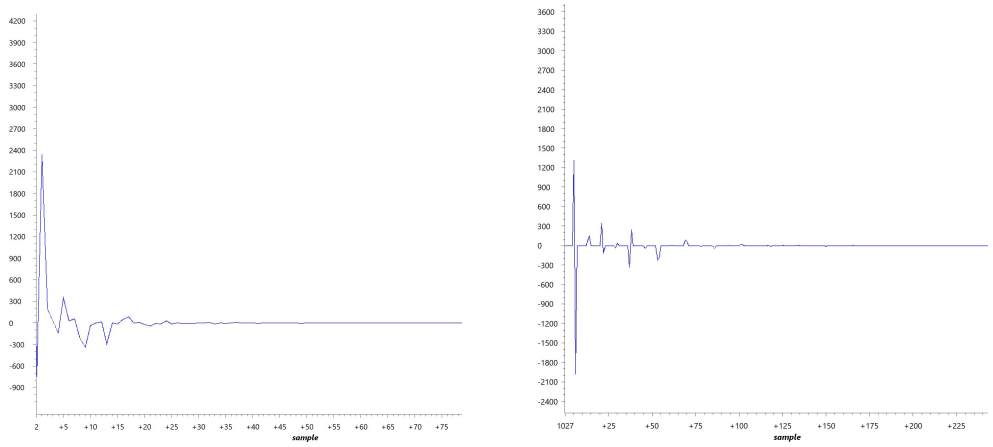
However, in the LEA RAM we also need to keep the Circular Buffer, where we store circularly the output values of the ADC, which is made of 516 Words and so occupies 1 kB, and the High-Pass FIR Filter result vector, called *result*, which takes for another 512 Bytes. Moreover, the FFT can be applied only on vectors that have a length which is a power of two, so we can't resort to a 3.5 Kb $fftBuffer$ variable.

After performing several attempts, we discover that the maximum length for the $fftBuffer$ is 1024 Words, for a total of 4 waveform periods, corresponding to 2 kB. This was made possible also by accurately studying the RAM locations assigned to the LEA RAM and available to the user and manually setting the allocated memory locations for the LEA RAM variables of *Circular Buffer*, *result* and *fftBuffer*.

```
1  /* ADC Output Register */
2  #pragma LOCATION(circularBuffer, 0x002C00)
3  _q15 circularBuffer[2* FIR_LENGTH];
4
5  /* FFT Input and Result */
6  #pragma LOCATION(fftBuffer, 0x003000)
7  _q15 fftBuffer[4*FIR_LENGTH];
8
9  /* Filter result */
10 #pragma LOCATION(result, 0x003800)
11 _q15 result[FIR_LENGTH];
```

We can now perform the FFT on the *fftBuffer* vector, which now includes a total of 4 signal periods that always start from the same point to obtain coherent results.

The spectrum resulting from the FFT has a much higher frequency resolution, as we can easily see in Figure 87, also comparing it with the previous one.



(a) Single-Period FFT (b) Improved FFT

Figure 87: FFT Quality Comparison

However, since the actual appliance recognition will be performed anyway in cloud, it results to be better to transmit to it the entire isolated waveform section, in such a way to exploit its computational power also for the FFT.

9.1.2 FFT in Cloud

We advance now the hypothesis of performing the FFT in cloud by sending to it, instead of the 14 words (28 Bytes) of the composed harmonics vector of the signal, an entire wave period made of 256 words for a total of 512Bytes (0.5 Kb). Even though this may look like an heavy increase in the bandwidth occupation, the nowadays WiFi capabilities make this data transfer totally negligible either in bandwidth occupation and required time.

If we perform the FFT in cloud and send to it the entire isolated section of the waveform, we are able to exploit the computational potential of MATLAB, to properly scale and process the signals before proceeding with their frequency domain analysis and consequent classification.

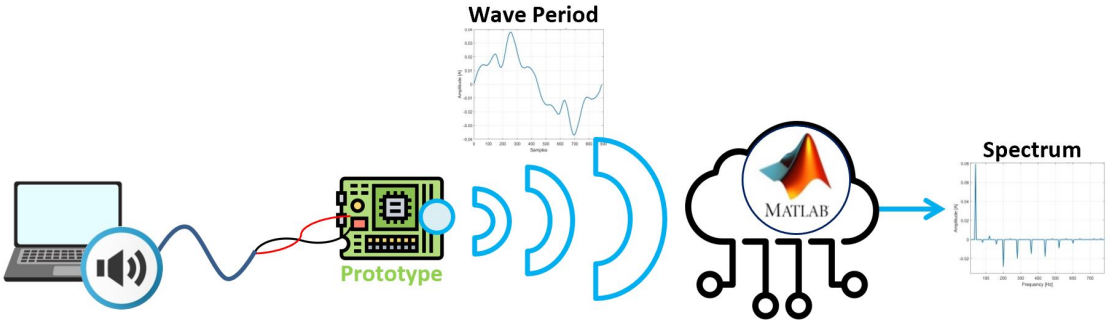


Figure 88: FFT in cloud configuration

9.2 Complete Prototype

We can appreciate the overall functioning of the architecture for the first explored prototype solution in Figure 89.

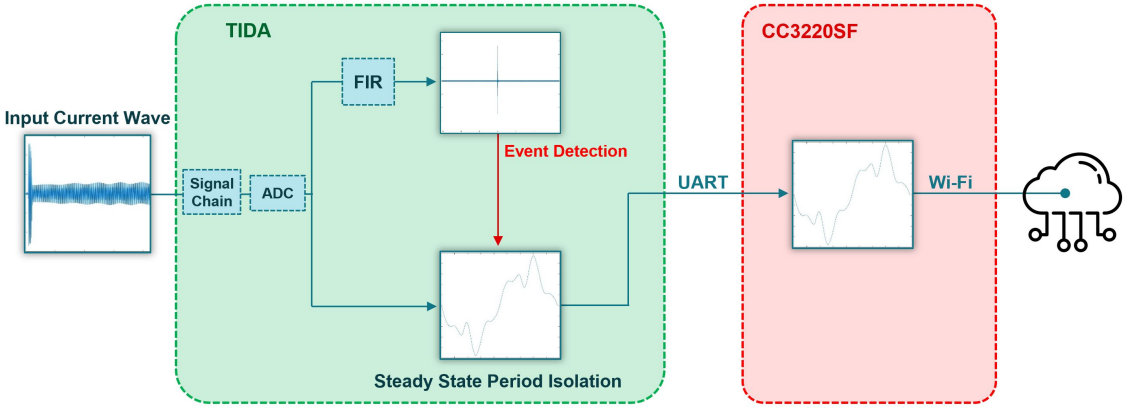


Figure 89: Complete Prototype Functioning

We can notice how the input current wave signal is fed through the TIDA's Signal Chain to the ADC of the MSP430. After the signal is acquired we have the realization, in parallel, of its FIR filtering for the event detection and, in case of an event, also of the isolation of a section of the waveform.
That same data is then transmitted via UART to the CC3220 which forwards it to the MQTT server and hence to the cloud server running MATLAB for the final appliance classification.

In Figure 90, we can instead observe the final prototype block diagram.

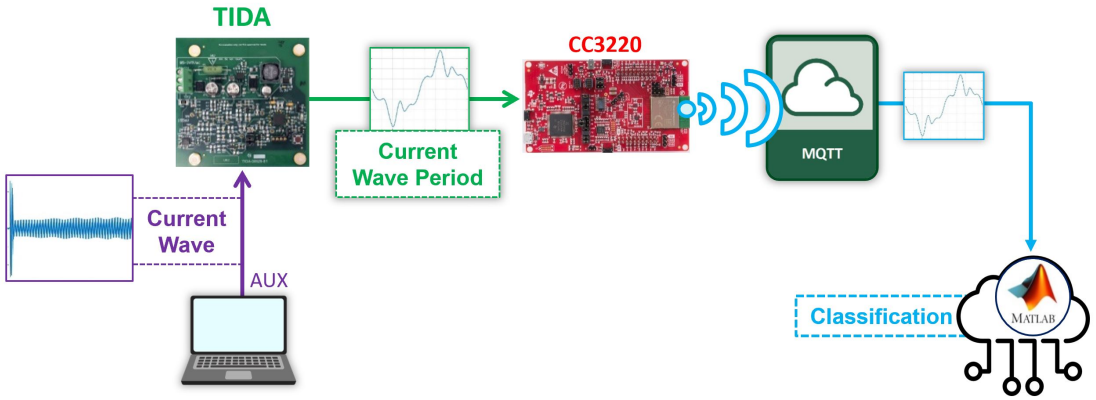


Figure 90: Complete Prototype Block Diagram

9.2.1 Classification in Cloud GUI

Similarly to what we did in the previous discussions, for the graphical visualization of the data that we were receiving via UART, we build now a MATLAB GUI (Graphic User Interface) to visualize at the cloud level the data that we are receiving from the prototype and the classification result, with the image of the resulting appliance. This GUI may resemble the appearances of a mobile application for the classification results communication and display.

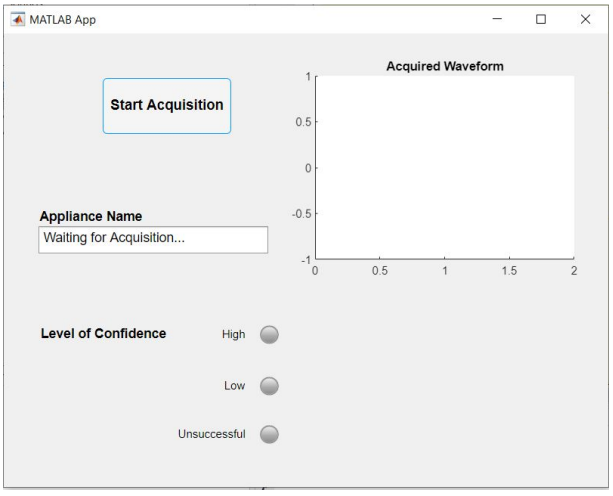
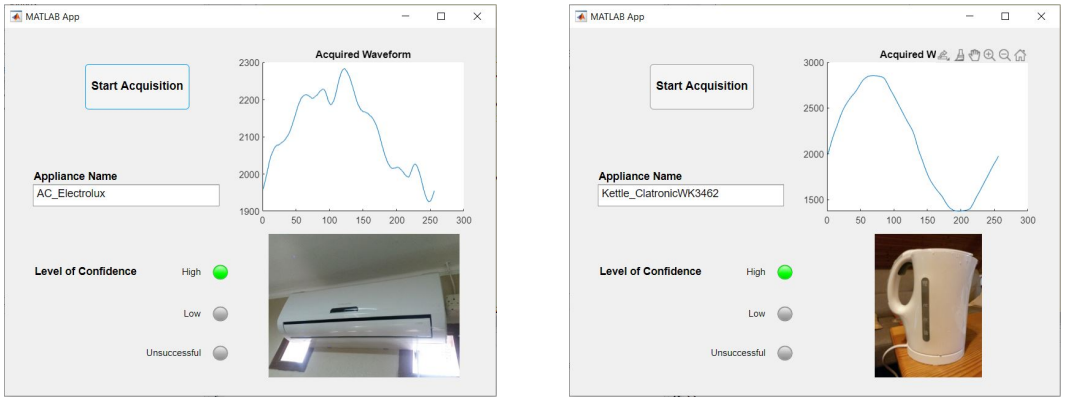


Figure 91: Classification in Cloud GUI



(a) (b)

Figure 92: GUI Output Examples

9.2.2 Classification in Cloud Results

Analyzing the results obtained with the classification in cloud configuration, we can identify, as its main features, the following aspects:

- High Recognition Accuracy $\sim 92\%$.
- Possibility to improve the recognition system by acting directly on cloud and exploiting its high computational power.
- New recognizable Appliances can be added without intervening on the prototype Software.

On the other hand, this configuration presents, as its main disadvantages, the following characteristics:

- Need for an Appliance Classification Service in Cloud.
- Underuse of the hardware involved in the prototype realization.

9.3 Appliance Classification in Edge

Considering the relative simplicity of the involved Neural Network, we evaluate the possibility to perform the FFT and the consequent spectrum identification and appliance classification directly in edge.

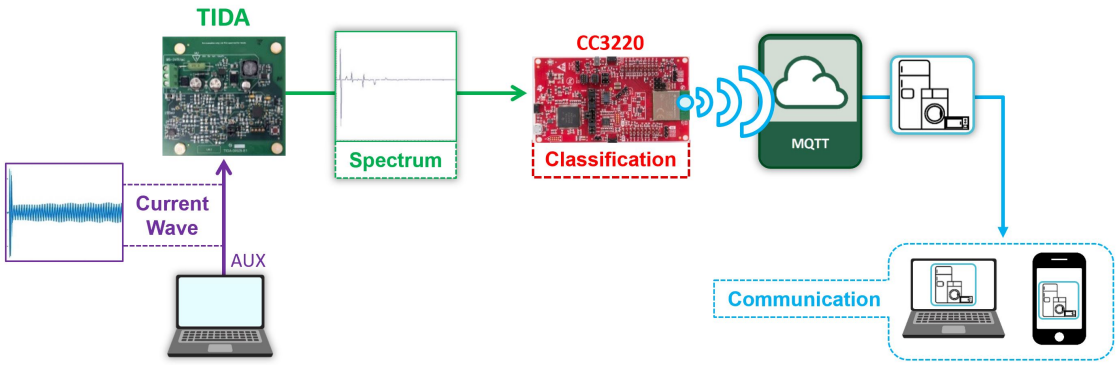


Figure 93: Block Diagram In-Edge Classification

The general idea is to perform the acquisition of the signal and its FFT on the TIDA-00929, with its MSP430FR5994, and to compute the absolute spectrum and isolate the spectral components of interest.

After that, we transmit via UART the values of the harmonics to the CC3220 with its Cortex-M4 and we perform the classification with the proper Neural Network reconstructed in the latter and, finally, we directly send to other devices the final classification outcome, without using any cloud.

9.3.1 Neural Network Structure

The neural network employed in the appliance classification algorithm is a single-layer perceptron, which means that it is only made of one input layer, one hidden one and one output one. For each layer we have to perform a matrix multiplication and sum with the weight and bias matrices, obtained at the training step. In addition, at the end of the Hidden and Output Layers we have two different activation functions.

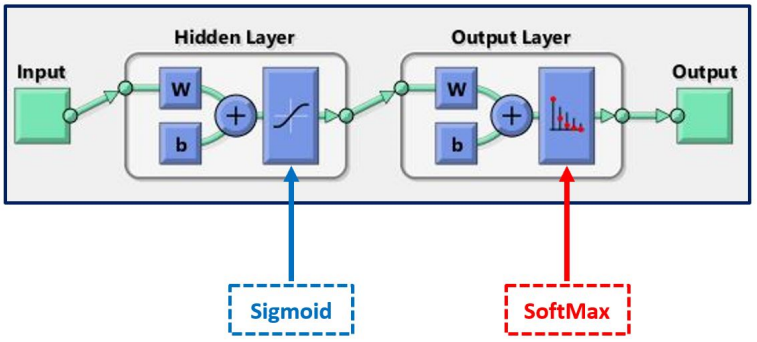


Figure 94: Neural Network Structure

In particular, between the Hidden Layer and the Output one we have a Sigmoid Function, which is used to normalise the values computed previously between zero and one, as can be observed by its transfer function in Figure 95.

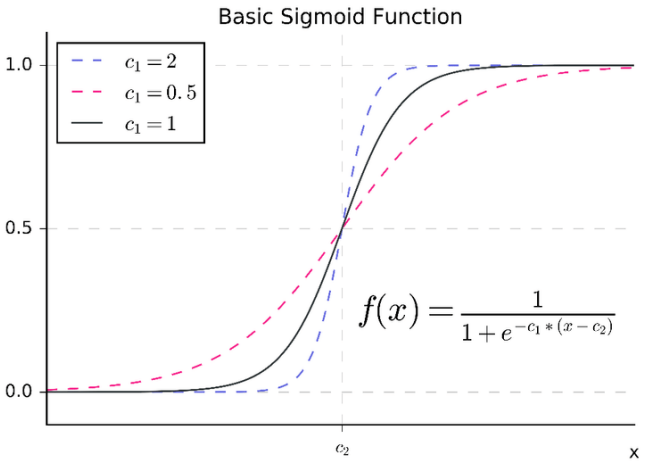


Figure 95: Sigmoid Activation Function

At the end of the Output Layer, instead, we have the SoftMax function, which normalise the values in the vector fed to it in such a way that the sum of all the elements will be one. Its expression and transfer function are pictured in Figure 96.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

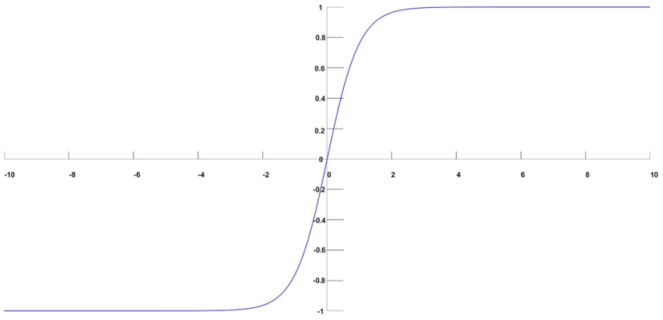


Figure 96: Transfer Function

In order to be able to implement the NN in edge, and therefore using embedded-C on the CC3220 board, we need to understand precisely all the operations performed by the neural network to go from the absolute harmonics' values to the classification of the appliance.

The first step is to model in Matlab a neural network, which is equivalent to the one that we want to realize in the micro-controller.

By analysing the operations performed by the feed-forward network, we observe that from the input, which is made of the 8 values of the first odd absolute harmonics, the following operations are performed:

1. Input Normalization → Sum and Multiplication of the input vector
2. Hidden Layer → Sum and Multiplication between matrices
3. Sigmoid Function
4. Output Layer → Sum and Multiplication between matrices
5. SoftMax Function
5. Classification

To understand properly the performed operations, we implement the Neural Network behaviour in Matlab, as reported in the following code snippet:

```
Input_layer= (harmonics_vector-x1_step1.xoffset).*x1_step1.gain + x1_step1.ymin
Hidden_Layer= 2 ./ (1 + exp(-2*(b1+IW1_1 * input_layer))) - 1
Output_Layer=   b2 +LW2_1 * Hidden_Layer

[val,classification]=max(Output_Layer);
classification
```

Since we are only interested in performing a classification of the appliance by selecting the corresponding output of the network that has the best fit with respect to the input vector fed to it, it is not necessary to implement an actual SoftMax function, but we can just select, as the classification output, the index of the maximum value from the Output Layer Vector.

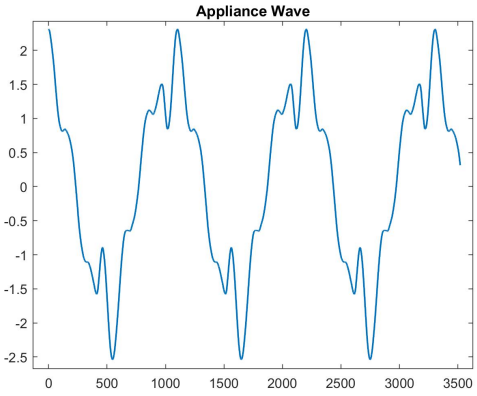
In fact, if we use the SoftMax function, we obtain a vector where are indicated the percentage of guess of each appliance for the analysed spectrum, where the appliance with the highest percentage is the final guess of the recognition mechanism.

However, if we simply compute the index of the max signed value of the Output Layer Vector we obtain the same result with a huge reduction in the computational cost of the network, which is extremely desirable since our purpose is to transfer this computations in edge, at Micro-Controller level.

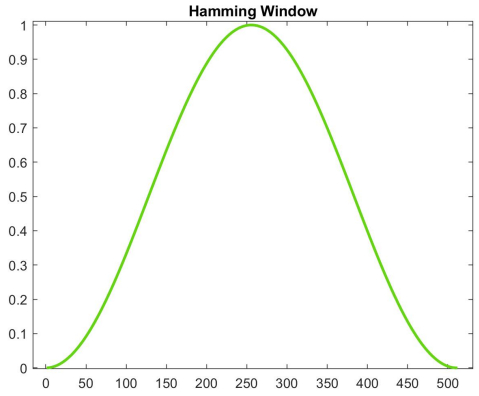
9.3.2 FFT at Local Hardware Level

In order to be able to classify the profile of the appliance waveform at local hardware level, we need first of all to perform an FFT accurately and coherently.

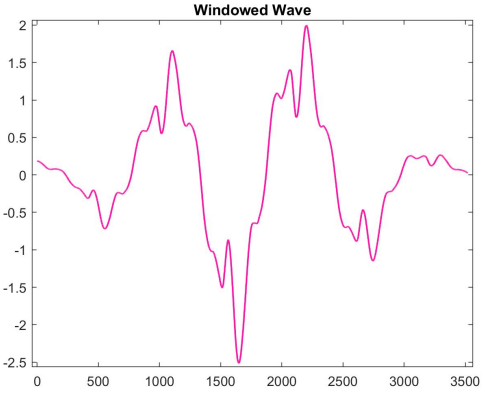
In addition to the considerations carried out in the previous sections, we evaluate two possible solutions: one of performing a plane *Discrete Fourier Transform* (DFT) on the data acquired, and the other one implying the multiplication of the waveform with an appropriate window before performing the frequency domain analysis. The window explored is the Hamming one, reported in Figure 97.



(a) Appliance Wave



(b) Window Signal



(c) Windowed Signal

Figure 97: Windowing of the Current Wave Signal

By using the technique of windowing the acquired signal before performing the FFT with a function like Hamming's, instead of the classic rectangular one, in the final spectrum computed we reduce the weight of imperfections and discontinuities that are inevitably present at the edges of the non-infinite section of the signal of which we perform the frequency analysis.

This operation results into an increase in the Spurious Free Dynamic Range (SFDR) of the spectrum and therefore in an overall enhancing of the harmonic components of interest.

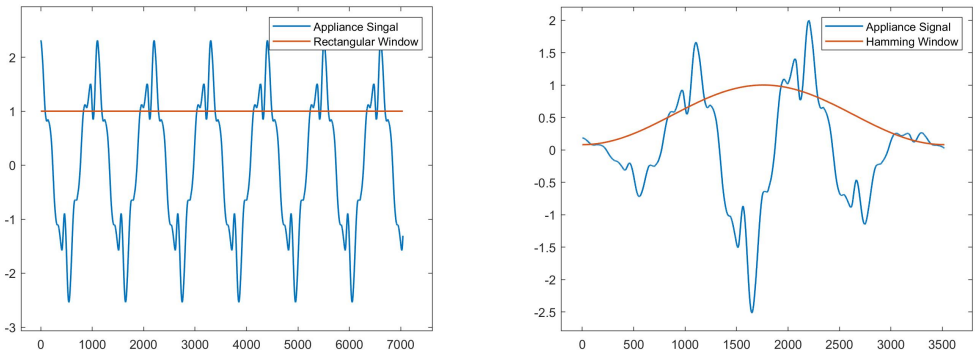
However, since we are performing the windowing and the FFT in the MSP430FR5994, we need to take into account its consequent memory limitations.

In particular, since those two operations, that require a fairly large computational effort, necessarily require the use of the LEA (Low Energy Accelerator) to be realized in an acceptable amount of time, we need to keep the data employed in the LEA-RAM, which is made of 4 Kb.

Therefore, when we evaluate the possibility of using a windowing of the acquired signal, we encounter a trade-off. In fact, if we need to save the Hamming Window in the LEA-RAM, then we will not have enough memory left to fit the 1024 signal samples on which, otherwise, we would perform the FFT. The two possible operative solutions analysed come down to performing:

1. FFT on more periods with Rectangular windowing.

2. FFT on less periods with Hamming windowing.



(a) Case 1 (b) Case 2

Figure 98: Windowing Trade-Off

After analysing the outcome of the FFT in the two cases, for numerous appliances measurements, since we are interested in the peak values of the absolute spectrum harmonic components, the increase in the Spurious Free Dynamic Range caused by the Hamming windowing of the signal does not provide any notable improvement in the collected data, while the reduction in the period exploited for the FFT that we encounter in this case leads to a significant worsening of the spectral outcome, as can be noticed in Figures 99 and 100.

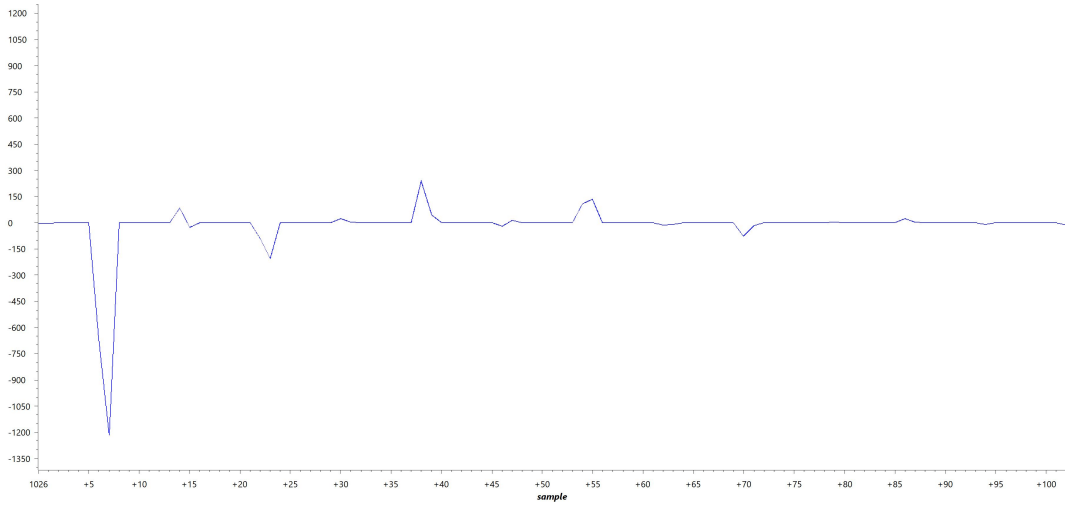


Figure 99: Spectrum with Rectangular Window

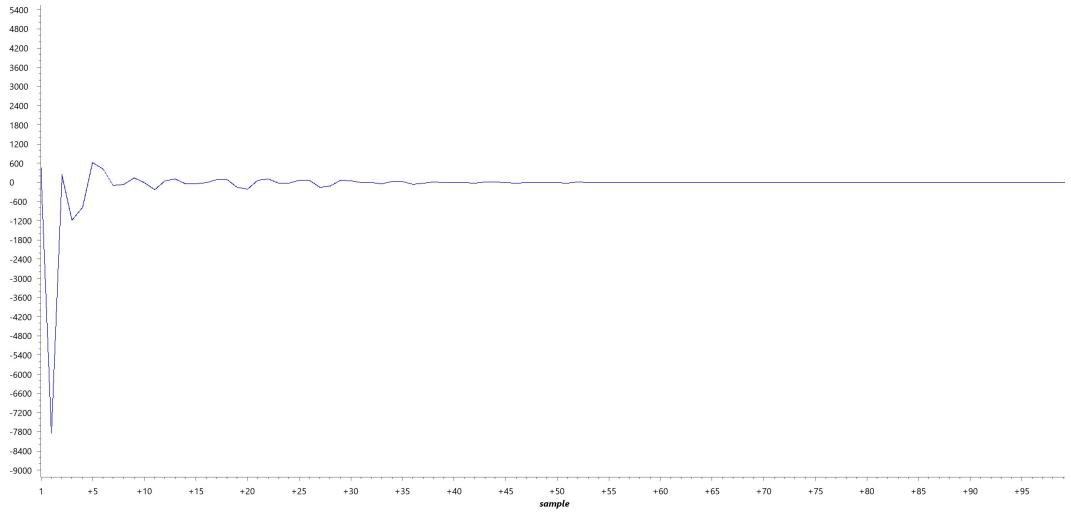


Figure 100: Spectrum with Hamming Window

9.3.3 Absolute Spectrum Computation

Now that we have decided to exploit an higher number of periods but a rectangular windowing for the frequency analysis, we need to compute the absolute spectrum of the signal from the resulting FFT and to isolate in it the values of the harmonics component of interest.

However, the computation of the absolute spectrum is not as trivial as it was on Matlab when we were performing the FFT in Matlab.

To perform the FFT we can rely on the DSP Library functions provided by the Software Development Kit of the MSP430FR994, and use the predefined function $msp\_fft\_auto\_q15$ which is capable of computing the auto-scaled FFT on a vector of Q15 values.

Since we are performing a DFT, in the output provided are present both the Real and Imaginary spectra merged in the same output vector in which the data of the two results are arranged in an alternated sequence, as can be seen in Figure 101.

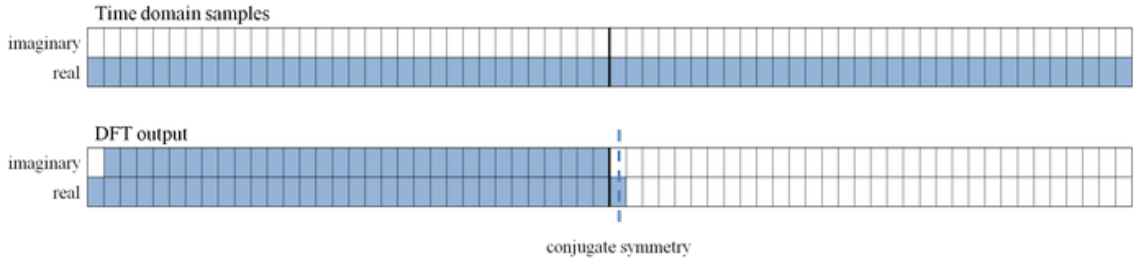


Figure 101: DFT output data arrangement

Therefore, to separate the Real and Imaginary components of the spectrum we need to divide, in the output vector, the values with even and odd indices as shown in the following code snippet:

```
index_imag=0;
index_real=0;

for(kk=1;kk<512;kk++){
    if(kk%2==0){//IMAGINARY
        result_fft[512+index_imag]=result_fft[kk];
        index_imag++;
    }
    else{        //REAL
        result_fft[index_real]=result_fft[kk];
        index_real++;
    }
}
```

After that, from the Real and Imaginary components of the spectrum we can simply compute the absolute spectrum as the square root of the sum of the two values squared.

The use of the absolute spectrum, even though it results into an halving of the usable data for the classification, makes the algorithm robust against phase variations in the isolation of the acquired signal.

For what concerns the isolation of the values of spectral components, we identify the index of fundamental harmonic in the vector of the spectrum by using a *max* function and then we identify the values of the rest of the harmonic content in the spectrum vector as multiples of the index of the fundamental.

9.3.4 Neural Network in the CC3220

In order to transfer the Neural Network in edge and, in particular, in the CC3220, first of all we need to compute the proper values of the Weight and Bias matrices that are used respectively in matrix multiplications and sums in its different layers. We need to collect a reliable on-field dataset of the harmonic components from the 10 available measurements of all the considered appliances and then proceed with the training phase of the perceptron.
Therefore, in this phase we use the CC3220 to forward the spectrum received via UART from the MSP430 to our PC running Matlab via Wi-Fi, where we can collect and use the spectrum for the actual training of the NN.

Successively, we can transfer the computed matrices in the CC3220 that will receive the harmonic components of interest from the MSP430 and will perform the same operations of the Neural Network and recognize the analyzed appliance. The final classification result is then transmitted through the same MQTT Broker mechanism described in Section 6.1 to all the devices connected to that broker and subscribed to the specific topic chosen for the communication.

However, for what concerns the two activation functions present in the Neural Network, while the Sigmoid Function has been realized as expected in the CC3220, the SoftMax was actually substituted by an evaluation of the index of the maximum value in the vector that would have been fed to it, since the normalization provided by the SoftMax activation function is not needed to obtain the only classification result.

Finally, in this way, we can easy transmit the classification result directly to the user and/or to the Energy Provider.

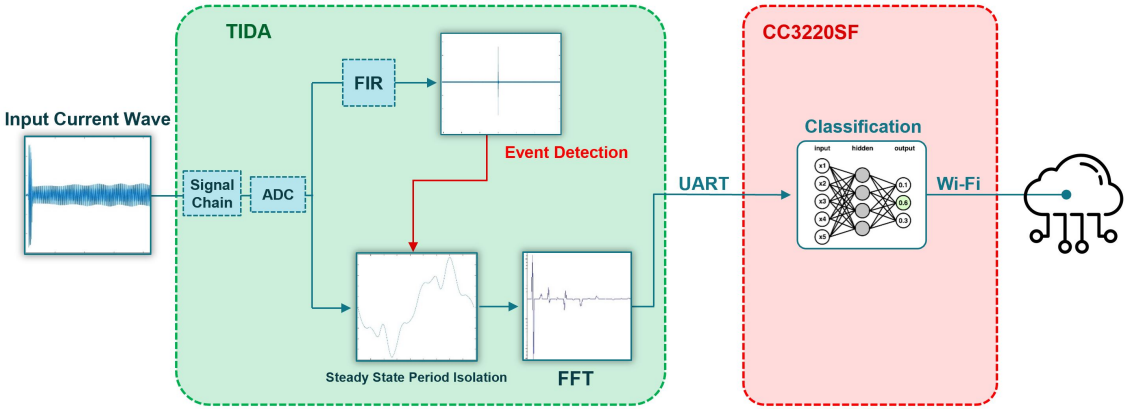
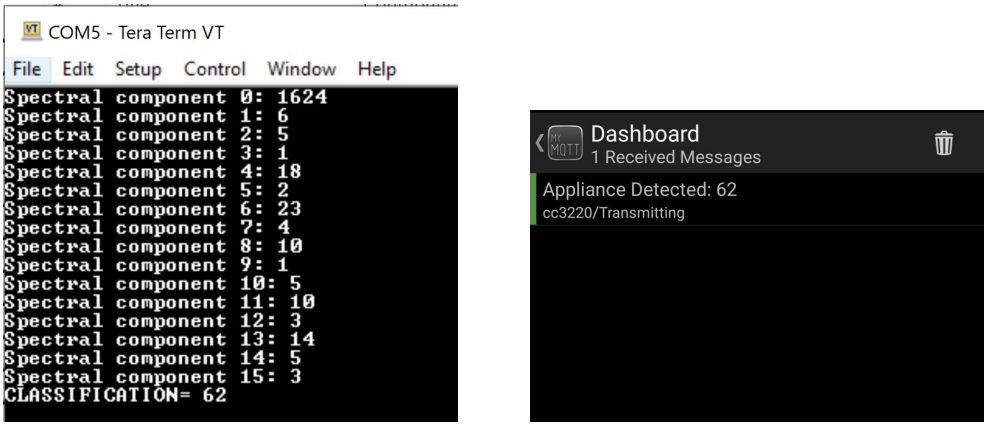


Figure 102: Edge-Class. Prototype Functioning

In Figure 103 we can observe an example of data that can be transmitted via Wi-Fi. In (a) the entire harmonic content and the final classification result, here actually sent via USB and displayed with Tera Term, and in (b) an example of the classification outcome that can be received on the user's Smartphone.



(a) Tera Term Tx Example (b) Smartphone Data Delivery

Figure 103: Example of data delivered to Other devices

9.3.5 Classification in Edge Results

Analyzing the main features of the classification in edge configuration and its results, we can highlight as its positive characteristics that:

- Good Accuracy Level $\sim 90\%$
- No need for an Appliance Classification Service in Cloud
- More adequate use of the Hardware provided

On the other hand, we can identify as the main disadvantage of this solution the following aspect:

- Software Updates needed to add new recognizable appliances or to improve the recognition mechanism

9.4 Further Developments

The study and the prototype realized and here discussed are not meant to be considered as an arriving point for this technologies, but its purpose is to lay the foundations for future analysis, in which several aspects of the project can be improved and optimized, going, for each of them, into details.

In fact, starting from the researches conducted so far, we identify as some of the basic characteristics of interest for further studies, the following ones:

- Improvement of the capability of disaggregation of multiple appliances overlapped, obtaining more consistent results.
- Realization of a Non-Event Based NILM, with a continuous monitoring of the current spectrum and its variations, with the relative estimation of the computational power and of the hardware required.
- Optimization of the balancing among the operations performed in Cloud and the ones realized at local hardware level.
- Capability of classifying also other classes of appliances in addition to the *On/Off* ones.
- Modeling of a single custom-PCB for the realization of the entire prototype, possibly exploiting a single Micro-Controller.
- Study and optimization of the Cost-Effectiveness trade-off for involvement of different categories processors.
- Development of a Mobile Application for an easier visualization and deliverance of the outcomes of the prototype.

10 Code Appendix

10.1 C Code

10.1.1 MSP-EXP430FR5994 Writing on USB

```
#include <msp430.h>

volatile unsigned char RXData = 0;
volatile unsigned char TXData = 'a';
int k;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                 // Stop watchdog

    // Configure GPIO
    P1OUT &= ~BIT0;                           // Clear P1.0 output latch
    P1DIR |= BIT0;                            // For LED on P1.0

    P2SEL1 |= (BIT0 | BIT1); // USCI_A0 UART operation
    P2SEL0 &= ~(BIT0 | BIT1);

    // Disable the GPIO power-on default high-impedance mode to activate
    // previously configured port settings
    PM5CTL0 &= ~LOCKLPM5;

    CSCTL0_H = CSKEY_H;                       // Unlock CS registers
    CSCTL1 = DCOFSEL_3 | DCORSEL;             // Set DCO to 8MHz
    CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
    CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;     // Set all dividers
    CSCTL0_H = 0;


    UCA0CTLW0 = UCSWRST;                      // Put eUSCI in reset
    UCA0CTLW0 |= UCSSEL__SMCLK;               // CLK = SMCLK
    // Baud Rate calculation
    // 8000000/(16*9600) = 52.083
    // Fractional portion = 0.083
    // User's Guide Table 21-4: UCBRSx = 0x04
    // UCBRFx = int ( (52.083-52)*16) = 1
    UCA0BRW = 52;         //52                    // 8000000/16/9600
    UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
    UCA0CTLW0 &= ~UCSWRST;                    // Initialize eUSCI
    UCA0IE |= UCRXIE;


    // Configure USCI_A3 for UART mode
    UCA0CTLW0 = UCSWRST;                      // Put eUSCI in reset
    UCA0CTLW0 |= UCSSEL__SMCLK;               // CLK = SMCLK
    UCA0BRW = 52;         //52                    // 8000000/16/9600
    UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;                  // 1000000/115200 - I
                                              // UCBRSx value = 0xD6 (See UG)
    UCA0CTLW0 &= ~UCSWRST;                    // release from reset
    UCA0IE |= UCRXIE;                         // Enable USCI_A3 RX interrupt


    UCA0TXBUF = 'r';
```

```
53
54      __bis_SR_register(LPM3_bits | GIE);     // Enter LPM3, interrupts enabled
55      __no_operation();
56     while (1)
57      {
58          while(!(UCA0IFG & UCTXIFG));
59          P1OUT |= BIT0;
60          UCA0TXBUF = TXData;                 // Load data onto buffer
61          __bis_SR_register(LPM0_bits | GIE); // Enter LPM0, interrupts enabled
62      }
63  }
64
65
66  #if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
67  #pragma vector=EUSCI_A0_VECTOR
68  __interrupt void USCI_A0_ISR(void)
69  #elif defined(__GNUC__)
70  void __attribute__ ((interrupt(EUSCI_A0_VECTOR))) USCI_A0_ISR (void) //
71  #else
72  #error Compiler not supported!
73  #endif
74  {
75      switch(__even_in_range(UCA0IV, USCI_UART_UCTXCPTIFG))
76      {
77          case USCI_NONE: break;
78          case USCI_UART_UCRXIFG:
79               RXData= UCA0RXBUF;             // Read buffer
80              if(RXData != TXData)            // Check value
81              {
82                  P1OUT |= BIT0;              // If incorrect turn on P1.0
83                  while(1);                   // Trap CPU
84              }
85              TXData++;                      // increment data byte
86              __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0 on reti
87              break;
88          case USCI_UART_UCTXIFG: break;
89          case USCI_UART_UCSTTIFG: break;
90          case USCI_UART_UCTXCPTIFG: break;
91          default: break;
92      }
93  }
```

10.1.2 TIDA-00929 Transmitting Data

```
1   #include <msp430.h>
2   #include <stdio.h>
3
4   volatile unsigned char RXData = 0;
5   volatile unsigned char TXData = 'a';
6   int k;
7
8   int main(void)
9   {
10
```

```
11
12      // Stop watchdog timer
13      WDTCTL = WDTPW | WDTHOLD;
14
15      CSCTL0_H = 0;
16
17      // Initialize GPIO
18      PAOUT = 0x0000; PADIR = 0xFFFF;
19      PBOUT = 0x0000; PBDIR = 0xFFFF;
20      PCOUT = 0x0000; PCDIR = 0xFFFF;
21      PDOUT = 0x0000; PDDIR = 0xFFFF;
22      PJOUT = 0x0000; PJDIR = 0xFFFF;
23      PM5CTL0 &= ~LOCKLPM5;
24
25      // DEBUG: Set P4.5 low
26      P4OUT &= ~BIT5;
27      P4OUT ^= BIT5;
28      P4OUT ^= BIT5;
29      P1OUT |=BIT1; //Set P1.1 high
30
31      WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
32
33      // Configure GPIO
34      P2SEL1 |= (BIT0 | BIT1); // USCI_A0 UART operation
35      P2SEL0 &= ~(BIT0 | BIT1);
36
37
38      // Disable the GPIO power-on default high-impedance mode to activate
39      // previously configured port settings
40      PM5CTL0 &= ~LOCKLPM5;
41
42      // Startup clock system with max DCO setting ~8MHz
43      CSCTL0_H = CSKEY_H;                     // Unlock CS registers
44      CSCTL1 = DCOFSEL_3 | DCORSEL;           // Set DCO to 8MHz
45      CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
46      CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;   // Set all dividers
47      CSCTL0_H = 0;                           // Lock CS registers
48
49      // Configure USCI_A0 for UART mode
50      UCA0CTLW0 = UCSWRST;                    // Put eUSCI in reset
51      UCA0CTLW0 |= UCSSEL__SMCLK;             // CLK = SMCLK
52      // Baud Rate calculation
53      // 8000000/(16*9600) = 52.083
54      // Fractional portion = 0.083
55      // User's Guide Table 21-4: UCBRSx = 0x04
56      // UCBRFx = int ( (52.083-52)*16) = 1
57      UCA0BRW = 52;        //52                 // 8000000/16/9600
58      UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
59      UCA0CTLW0 &= ~UCSWRST;                  // Initialize eUSCI
60      UCA0IE |= UCRXIE;                       // Enable USCI_A0 RX interrupt
61
62      while(1){
63          P4OUT ^= BIT5;
64         UCA0TXBUF = 'a';
65          __delay_cycles(800000);
66          UCA0TXBUF = 'l';
67          __delay_cycles(800000);
68          UCA0TXBUF = 'b';
69          __delay_cycles(800000);
70          UCA0TXBUF = 'e';
```

```
71          __delay_cycles(8000000);
72          P4OUT^=BIT4;
73
74      }
75      __delay_cycles(20000);
76      while(1)
77      __bis_SR_register(LPM0_bits | GIE);     // Enter LPM3, interrupts enabled
78      __no_operation();                       // For debugger
79
80      return 0;
81  }
82
83  #if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
84  #pragma vector=EUSCI_A0_VECTOR
85  __interrupt void USCI_A0_ISR(void)
86  #elif defined(__GNUC__)
87  void __attribute__ ((interrupt(EUSCI_A0_VECTOR))) USCI_A0_ISR (void) //
88  #else
89  #error Compiler not supported!
90  #endif
91  {
92      switch(__even_in_range(UCA0IV, USCI_UART_UCTXCPTIFG))
93      {
94          case USCI_NONE: break;
95          case USCI_UART_UCRXIFG:
96              RXData = UCA0RXBUF;             // Read buffer
97              __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0 on reti
98              break;
99          case USCI_UART_UCTXIFG: break;
100         case USCI_UART_UCSTTIFG: break;
101         case USCI_UART_UCTXCPTIFG: break;
102         default: break;
103     }
104 }
```

10.1.3 MSP-EXP430FR5994 Receiving from TIDA and Sending to PC via USB

```
#include <msp430.h>

unsigned char RXData_tmp;RXData[100],TXData;
int index=0;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                 // Stop Watchdog

    PAOUT = 0x0000; PADIR = 0xFFFF;
    PBOUT = 0x0000; PBDIR = 0xFFFF;
    PCOUT = 0x0000; PCDIR = 0xFFFF;
    PDOUT = 0x0000; PDDIR = 0xFFFF;
    PJOUT = 0x0000; PJDIR = 0xFFFF;
    PM5CTL0 &= ~LOCKLPM5;

    // DEBUG: Set P4.5 low
    P1OUT &= ~BIT5;
    P1OUT ^= BIT5;
    P1OUT ^= BIT5;
    P1OUT |=BIT1; //Set P1.1 high
    // Configure GPIO

    P6SEL1 &= ~(BIT0 | BIT1);                 // USCI_A3 UART operation
    P6SEL0 |= BIT0 | BIT1;

    P2SEL1 |= (BIT0 | BIT1); // USCI_A0 UART operation
    P2SEL0 &= ~(BIT0 | BIT1);

    // Disable the GPIO power-on default high-impedance mode to activate
    // previously configured port settings
    PM5CTL0 &= ~LOCKLPM5;


    CSCTL0_H = CSKEY_H;                       // Unlock CS registers
    CSCTL1 = DCOFSEL_3 | DCORSEL;             // Set DCO to 8MHz
    CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
    CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;     // Set all dividers
    CSCTL0_H = 0;                             // Lock CS registers

    // Configure USCI_A3 for UART mode
    UCA3CTLW0 = UCSWRST;                      // Put eUSCI in reset
    UCA3CTLW0 |= UCSSEL__SMCLK;               // CLK = SMCLK
    UCA3BRW = 52;         //52                    // 8000000/16/9600
    UCA3MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
    UCA3CTLW0 &= ~UCSWRST;                    // Initialize eUSCI
    UCA3IE |= UCRXIE;                           // Enable USCI_A3 RX interrupt

    // Configure USCI_A0 for UART mode
    UCA0CTLW0 = UCSWRST;                      // Put eUSCI in reset
    UCA0CTLW0 |= UCSSEL__SMCLK;               // CLK = SMCLK
    UCA0BRW = 52;         //52                    // 8000000/16/9600
    UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
    UCA0CTLW0 &= ~UCSWRST;                    // Initialize eUSCI
    UCA0IE |= UCRXIE;                           // Enable USCI_A3 RX interrupt

```

```
  //  __bis_SR_register(LPM3_bits | GIE);     // Enter LPM3, interrupts enabled
  //  __no_operation();
   while (1)
    {

        __bis_SR_register(LPM0_bits | GIE); // Enter LPM0, interrupts enabled
    }
}

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=EUSCI_A3_VECTOR
__interrupt void USCI_A3_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(EUSCI_A3_VECTOR))) USCI_A3_ISR (void)
#else
#error Compiler not supported!
#endif
{

    switch(__even_in_range(UCA3IV, USCI_UART_UCTXCPTIFG))
    {
        case USCI_NONE: break;
        case USCI_UART_UCRXIFG:
         //   while(!(UCA3IFG&UCTXIFG));
            P1OUT^=BIT1;
            RXData_tmp= UCA3RXBUF;
            RXData[index%100]=RXData_tmp;
            //__delay_cycles(8000000); //wait 1 sec, we work at 8Mhz
            UCA0TXBUF = RXData[index%100];                  // Load data onto buf
            index++;
            while(!(UCTXIFG));

//            __no_operation();
            break;
        case USCI_UART_UCTXIFG: break;
        case USCI_UART_UCSTTIFG: break;
        case USCI_UART_UCTXCPTIFG: break;
        default: break;
    }
}
```

10.1.4 TIDA Event Detection and UART Acquired Wave Transmission

```
#include <msp430.h>
#include <math.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include "DSPLib.h"
#include "FIR_3.h"
//System parameters
#define SYSTEM_MCLK          16000000
#define SYSTEM_SMCLK         SYSTEM_MCLK
#define SAMPLE_LENGTH        256
#define SAMPLE_FREQUENCY     25600/////12800  i doubled the sampling frequency b
```

```
/* Filter parameters */
#define FIR_LENGTH      256
#define COEFF_LENGTH     sizeof(FILTER_COEFFS_EX1)/sizeof(FILTER_COEFFS_EX1[0])

#pragma LOCATION(circularBuffer, 0x002C00) //// it was 3000
////////////////////////////DSPLIB_DATA(circularBuffer,4)
_q15 circularBuffer[2* FIR_LENGTH];


/* Filter result */
#pragma LOCATION(fftBuffer, 0x003000) //// it was 3000
////////////////////////////DSPLIB_DATA(fftBuffer,4)
_q15 fftBuffer[2*FIR_LENGTH];

#pragma LOCATION(Data_To_Transmit, 0x003400) //// it was 3000
////////////////////////////DSPLIB_DATA(fftBuffer,4)
_q15 Data_To_Transmit[FIR_LENGTH];

////////////////////////////////DSPLIB_DATA(result,4)
#pragma LOCATION(result, 0x003800)
_q15 result[FIR_LENGTH];

/*Filter Data*/
//#pragma PERSISTENT(firdata) ////////////////////////////
//_q15 firdata[FIR_LENGTH] = {0}; ////////////////////////////////

msp_fir_q15_params firParams;
msp_fill_q15_params fillParams;
msp_max_q15_params firmaxParams;

DSPLIB_DATA(pulse_counts,4)
_q15 pulse_counts[8];

DSPLIB_DATA(FIR_peaks,4)
_q15 FIR_peaks[16];

volatile uint16_t cycle_counts = 0;
volatile uint16_t DataCopyPointer = 0;

volatile int16_t *previousBuffer;
volatile int16_t *activeBuffer;
volatile int16_t *nextBuffer;
volatile int16_t *firmax;

volatile uint16_t FilterIndex;
uint16_t firmaxIndex;
static const samples=0;
_q15 temp;
_q15 temp2;
static uint16_t caption;
uint8_t TX_Buffer;
uint16_t closest_value;
uint16_t i;
uint16_t kk;
uint16_t average;
 uint16_t IDLE;
 uint16_t IDLE_TIME;
 uint16_t IDLE_TIME_MAX;
 uint16_t ZERO_C;
 uint16_t ppp;
```

```
73  int main(void)
74  {
75      msp_status status;
76
77      // Stop watchdog timer
78      WDTCTL = WDTPW | WDTHOLD;
79
80      // Initialize clock settings
81      CSCTL0_H = CSKEY >> 8;
82  #if (SYSTEM_MCLK == 8000000)
83      CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;
84  #elif (SYSTEM_MCLK == 16000000)
85      FRCTL0 = FRCTLPW | NWAITS_1;
86      CSCTL3 = DIVA__4 | DIVS__4 | DIVM__4;
87      CSCTL1 = DCOFSEL_4 | DCORSEL; __delay_cycles(60);
88      CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;
89  #else
90  #error "Unsupported MCLK setting"
91  #endif
92      CSCTL0_H = 0;
93
94      // Initialize GPIO
95      PAOUT = 0x0000; PADIR = 0xFFFF;
96      PBOUT = 0x0000; PBDIR = 0xFFFF;
97      PCOUT = 0x0000; PCDIR = 0xFFFF;
98      PDOUT = 0x0000; PDDIR = 0xFFFF;
99      PJOUT = 0x0000; PJDIR = 0xFFFF;
100     PM5CTL0 &= ~LOCKLPM5;
101
102     // DEBUG: Set P4.5 low
103     P4OUT &= ~BIT5;
104     P4OUT ^= BIT5;
105     P4OUT ^= BIT5;
106     P1OUT |=BIT1; //Set P1.1 high
107     // Configure ADC: P1.0/A0
108     P1SEL0 |= BIT0;
109     P1SEL1 |= BIT0;
110     //Configure P1.2 for comparator input C2
111     P1SEL0 |= BIT2;
112     P1SEL1 |= BIT2;
113     //configure P2.0 to be input TB0CLK from the external comparator output
114     //this is enable only if external comparator is used
115
116     //P2DIR &= ~BIT0;
117     //P2SEL1 |= BIT0;
118     //P2SEL0 |= BIT0;
119     P1DIR &= ~BIT6; //configure P1.6 as input since it is connected to the compo
120
121
122     PM5CTL0 &= ~LOCKLPM5;
123
124     CSCTL0_H = CSKEY_H;                     // Unlock CS registers
125     CSCTL1 = DCOFSEL_3 | DCORSEL;           // Set DCO to 8MHz
126     CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
127     CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;   // Set all dividers
128     CSCTL0_H = 0;                           // Lock CS registers
129
130     // DEBUG: Set P4.5 low
131     P4OUT &= ~BIT5;
132     P4OUT ^= BIT5;
```

```
133     P4OUT ^= BIT5;
134     P1OUT |=BIT1; //Set P1.1 high
135
136     // Configure GPIO
137     P2SEL1 |= (BIT0 | BIT1); // USCI_A0 UART operation
138     P2SEL0 &= ~(BIT0 | BIT1);
139
140     // Configure USCI_A0 for UART mode
141     UCA0CTLW0 = UCSWRST;                      // Put eUSCI in reset
142     UCA0CTLW0 |= UCSSEL__SMCLK;               // CLK = SMCLK
143     UCA0BRW = 52;         //52                    // 8000000/16/9600
144     UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
145     UCA0CTLW0 &= ~UCSWRST;                    // Initialize eUSCI
146     UCA0IE |= UCRXIE;                         // Enable USCI_A0 RX interrupt
147
148     __delay_cycles(6000);
149
150     // Configure TA2 as counter with COUT as input clock, enable overflow inter
151     TA2CTL = TASSEL__TACLK | MC__CONTINUOUS | TACLR | TAIE;
152     // Configure TB0 as counter with TB0CLK/P2.0 as input clock, enable overflo
153     //THIS IS USED IF EXTERNAL COMPARATOR IS USED
154     //TB0CTL = TASSEL__0 | MC__CONTINUOUS | TBCLR | TBIE;
155
156     // Configure COMP and P1.2 as C2 input
157 //    P1SEL0 |= BIT2;
158 //    P1SEL1 |= BIT2;
159
160     // Configure TA1 timer to provide interval
161     TA1CCR0 = (SYSTEM_SMCLK / SAMPLE_FREQUENCY) - 1;
162     TA1CCR1 = (SYSTEM_SMCLK / SAMPLE_FREQUENCY / 2) - 1;
163     TA1CCTL1 = OUTMOD_3;
164     TA1CTL = TASSEL__SMCLK | ID__1;
165
166     // Configure ADC12 for sampling
167     ADC12CTL0 = ADC12SHT0_0 | ADC12SHT1_0 | ADC12ON;
168     ADC12CTL1 = ADC12SHS_4 | ADC12CONSEQ_2 | ADC12SHP;
169     ADC12CTL2 = ADC12RES_2 | ADC12DF_0; //ADC12DF_0: Unsigned 12-bit read, ADC12
170     ADC12MCTL0 |= ADC12INCH_0;
171     ADC12CTL0 |= ADC12ENC | ADC12SC;
172
173     // Configure DMA channel 0 to transfer ADC samples
174     __data20_write_long((uintptr_t)&DMA0SZ, (uint16_t)SAMPLE_LENGTH); //
175     __data20_write_long((uintptr_t)&DMA0SA, (uintptr_t)&ADC12MEM0);
176     __data20_write_long((uintptr_t)&DMA0DA, (uintptr_t)circularBuffer);
177     DMACTL0 = DMA0TSEL__ADC12IFG;
178     DMA0CTL = DMADT_0 | DMASRCINCR_0 | DMADSTINCR_3 | DMAIE | DMAEN;
179
180     // Initialize DMA buffer pointers
181     previousBuffer =&circularBuffer[SAMPLE_LENGTH]; ////////////////////&circula
182     activeBuffer = &circularBuffer[0];
183     nextBuffer = previousBuffer;
184
185     //Initialize FFT variables
186     uint16_t shift;
187     msp_cmplx_fft_q15_params fftParams;
188
189     /* Initialize the fft parameter structure. */
190     fftParams.length = 512; ////////////////////////IT WAS 128 ///////////////
191     fftParams.bitReverse = 1;
192     fftParams.twiddleTable = msp_cmplx_twiddle_table_512_q15; ////////////////
```

```
193
194     // Initialize FIR pointers
195     FilterIndex = 2*FIR_LENGTH-COEFF_LENGTH;
196
197     /* Initialize the FIR parameter structure. */
198     firParams.length = FIR_LENGTH;
199     firParams.tapLength = COEFF_LENGTH;
200     firParams.coeffs = FILTER_COEFFS_EX1;
201     firParams.enableCircularBuffer = true;
202
203     /* Zero initialize FIR input for use with circular buffer. */
204     fillParams.length = 2*FIR_LENGTH;
205     fillParams.value = 0;
206     status = msp_fill_q15(&fillParams, circularBuffer);
207     msp_checkStatus(status);
208
209     /*Max Value function initialize */
210     firmaxParams.length = FIR_LENGTH;
211
212     _q15 q15MaxVector;
213     uint16_t uint16MaxIndex;
214     uint16MaxIndex=0;
215
216
217     //Initialize general variables
218     closest_value=0;
219     kk=0;
220     ppp=0;
221     ZERO_C=0;
222     average=0;
223     i=0;
224     IDLE=0;
225     IDLE_TIME=0;
226     IDLE_TIME_MAX= 512;
227    // temp=FIR_LENGTH;
228     temp=0;
229     caption=0;
230     DataCopyPointer=0;
231   // Enable interrupts and start conversion
232     __enable_interrupt();
233     TA1CTL |= MC__UP | TACLR;
234
235     while(1) {
236         cycle_counts = 0;
237         temp2=temp;
238         temp=0;
239         if (caption==1)
240             {
241             caption=0;
242             //memcpy(&fftBuffer, circularBuffer, FIR_LENGTH * sizeof(_q15) );
243             memcpy(&result, circularBuffer, FIR_LENGTH * sizeof(_q15) );
244
245             status = msp_max_q15(&firmaxParams,result ,&q15MaxVector,&uint16MaxIn
246             msp_checkStatus(status);
247
248             average=0;
249             for(kk=0; kk<FIR_LENGTH; kk++){
250                 average += result[kk]/FIR_LENGTH;
251             }
252
```

```
253             closest_value=1000;
254             for(kk=0; kk<FIR_LENGTH; kk++){
255                if((average - result[kk] <closest_value) && (average - result[kk]
256                    ZERO_C=kk;
257                    closest_value=average - result[kk];
258                }
259             }
260
261             memcpy(&fftBuffer, result, FIR_LENGTH * sizeof(_q15) );
262             memcpy(&fftBuffer[FIR_LENGTH], fftBuffer, FIR_LENGTH * sizeof(_q15) )
263             memcpy(&fftBuffer[2*FIR_LENGTH], fftBuffer, FIR_LENGTH * sizeof(_q15)
264
265             memcpy(&Data_To_Transmit, &fftBuffer[ZERO_C], (FIR_LENGTH) * sizeof(_
266
267            // memcpy(&fftBuffer[FIR_LENGTH], &fftBuffer, FIR_LENGTH * sizeof(_q
268            // memcpy(&fftBuffer[2*FIR_LENGTH], &fftBuffer, 2*FIR_LENGTH * sizeo
269
270             for(kk=0;kk<256;kk++){
271                 //P4OUT ^= BIT5;
272              for(ppp=0;ppp<2;ppp++){
273
274                  if(ppp==1){
275                       TX_Buffer = (uint8_t)(Data_To_Transmit[kk] >> 8);
276                  }
277                  else if(ppp==0){
278                      TX_Buffer = (uint8_t)(Data_To_Transmit[kk]& 0x00FF);
279                  }
280                       UCA0TXBUF =TX_Buffer;
281                       __delay_cycles(10000);
282                 }
283
284             }
285
286            // FFT !
287             //status = msp_cmplx_fft_auto_q15(&fftParams,fftBuffer, &shift);// I
288            // msp_checkStatus(status);
289             P4OUT^=BIT4;
290
291             }
292
293
294
295
296     while(cycle_counts<8*8*4)
297     {
298         // Enter LPM0 and wait for DMA ISR
299         __bis_SR_register(LPM0);
300
301
302         P4OUT ^= BIT5;
303
304         ///AVOID DETECTION AT THE END
305         if(IDLE==1){
306             IDLE_TIME++;
307             if(IDLE_TIME >= IDLE_TIME_MAX){
308                 IDLE=0;
309                 IDLE_TIME=0;
310             }
311         }
312
```

```
313         // Run FIR filter
314         status = msp_fir_q15(&firParams, &circularBuffer[FilterIndex], &result[sa
315         msp_checkStatus(status);
316         FilterIndex ^= FIR_LENGTH;
317        // memcpy(&firdata[DataCopyPointer], &result[samples], FIR_LENGTH*2);
318         //DataCopyPointer += FIR_LENGTH;
319 // Find max in result
320         status = msp_max_q15(&firmaxParams,result ,&q15MaxVector,&uint16MaxIndex)
321         msp_checkStatus(status);
322 //Compare max with previous (with temp and temp2 we avoid to detect as event th
323         if (((q15MaxVector>1.3*temp2 &&temp2!=0) ||(q15MaxVector>1.3*temp &&temp!
324             caption+=1;
325             DataCopyPointer++;
326             IDLE=1;
327         /* Perform complex FFT with auto scaling */
328            // memcpy(fftBuffer, result, FIR_LENGTH * sizeof(_q15) );
329            // status = msp_cmplx_fft_auto_q15(&fftParams,fftBuffer, &shift);
330            // msp_checkStatus(status);
331             P4OUT ^= BIT4;  //LED
332         }
333
334         if (q15MaxVector > temp)
335             temp=q15MaxVector;
336
337         cycle_counts++;
338         // DEBUG: Toggle P4.5
339         P4OUT ^= BIT5;
340
341
342         }
343         //P4OUT ^= BIT4;
344         i+=1;
345 }
346 }
347     // DMA interrupt service routine
348 #pragma vector = DMA_VECTOR
349 __interrupt void DmaIsr(void)
350 {
351     switch(__even_in_range(DMAIV, DMAIV_DMA0IFG)) {
352     case DMAIV_NONE: break;
353     case DMAIV_DMA0IFG:
354         // Setup next transfer
355         __data20_write_long((uintptr_t)&DMA0DA, (uintptr_t)nextBuffer);
356         DMA0CTL |= DMAEN;
357
358         // Switch buffer pointers
359         previousBuffer = activeBuffer;
360         activeBuffer = nextBuffer;
361         nextBuffer = previousBuffer;
362
363         // Exit from LPM0
364         __bic_SR_register_on_exit(LPM0_bits);
365         break;
366     default: break;
367     }
368 }
```

10.2 TIDA Event detection with FFT and spectrum TX

```
#include <msp430.h>
#include <math.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <complex.h>
#include "DSPLib.h"
#include "FIR_Bandpass.h"
//System parameters
#define SYSTEM_MCLK          16000000 //16
#define SYSTEM_SMCLK         SYSTEM_MCLK
#define SAMPLE_LENGTH        256
#define SAMPLE_FREQUENCY     12800 //25600//12800
/* Filter parameters */
#define FIR_LENGTH       256
#define COEFF_LENGTH      sizeof(FILTER_COEFFS_EX1)/sizeof(FILTER_COEFFS_EX1[0])



#pragma LOCATION(circularBuffer, 0x002C00)
//DSPLIB_DATA(circularBuffer,4)
_q15 circularBuffer[2* FIR_LENGTH];
//DSPLIB_DATA(circularBuffer,4)

//DSPLIB_DATA(window,4)
//_q15 window[FIR_LENGTH];

/* Filter result */
//#pragma LOCATION(fftBuffer, 0x003000)


DSPLIB_DATA(result,4)
//#pragma LOCATION(result, 0x003000)
_q15 result[4*FIR_LENGTH];


//#pragma LOCATION(Data_To_Transmit, 0x003900)//// it was 3000
DSPLIB_DATA(Data_To_Transmit,4)
_q15 Data_To_Transmit[16];

DSPLIB_DATA(spectrum_value,4)
_q15 spectrum_value[6];


msp_fir_q15_params firParams;
msp_fill_q15_params fillParams;
msp_max_q15_params firmaxParams;
msp_max_q15_params tmpParams;
msp_min_q15_params firminParams;

DSPLIB_DATA(pulse_counts,4)
_q15 pulse_counts[8];

DSPLIB_DATA(FIR_peaks,4)
_q15 FIR_peaks[16];

volatile uint16_t cycle_counts = 0;
```

```
58  volatile uint16_t DataCopyPointer = 0;
59
60  volatile int16_t *previousBuffer;
61  volatile int16_t *activeBuffer;
62  volatile int16_t *nextBuffer;
63  volatile int16_t *firmax;
64
65  volatile uint16_t FilterIndex;
66  uint16_t firmaxIndex;
67  static const int samples=0;
68  _q15 temp;
69  _q15 temp2;
70  uint16_t fundamental_index;
71  _q15 delta_previous;
72  _q15 delta;
73  static uint16_t caption;
74  uint16_t i;
75  uint16_t kk;
76  uint16_t index_imag, index_real;
77  uint16_t average;
78   uint16_t IDLE;
79   uint16_t IDLE_TIME;
80   uint16_t IDLE_TIME_MAX;
81   uint16_t ZERO_C;
82   uint16_t ppp;
83  uint8_t TX_Buffer;
84  uint16_t redundancy;
85  uint16_t spikes_avoid;
86
87  int main(void)
88  {
89      msp_status status;
90
91      //initHamming();
92
93      // Stop watchdog timer
94      WDTCTL = WDTPW | WDTHOLD;
95
96      // Initialize clock settings
97      CSCTL0_H = CSKEY >> 8;
98  #if (SYSTEM_MCLK == 8000000)
99      CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;
100 #elif (SYSTEM_MCLK == 16000000)
101     FRCTL0 = FRCTLPW | NWAITS_1;
102     CSCTL3 = DIVA__4 | DIVS__4 | DIVM__4;
103     CSCTL1 = DCOFSEL_4 | DCORSEL; __delay_cycles(60);
104     CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;
105 #else
106 #error "Unsupported MCLK setting"
107 #endif
108
109     CSCTL0_H = 0;
110
111     // Initialize GPIO
112     PAOUT = 0x0000; PADIR = 0xFFFF;
113     PBOUT = 0x0000; PBDIR = 0xFFFF;
114     PCOUT = 0x0000; PCDIR = 0xFFFF;
115     PDOUT = 0x0000; PDDIR = 0xFFFF;
116     PJOUT = 0x0000; PJDIR = 0xFFFF;
117     PM5CTL0 &= ~LOCKLPM5;
118
```

```
119     // DEBUG: Set P4.5 low
120     P4OUT &= ~BIT5;
121     P4OUT ^= BIT5;
122     P4OUT ^= BIT5;
123     P1OUT |=BIT1; //Set P1.1 high
124     // Configure ADC: P1.0/A0
125     P1SEL0 |= BIT0;
126     P1SEL1 |= BIT0;
127     //Configure P1.2 for comparator input C2
128     P1SEL0 |= BIT2;
129     P1SEL1 |= BIT2;
130     //configure P2.0 to be input TB0CLK from the external comparator output
131     //this is enable only if external comparator is used
132     P2DIR &= ~BIT0;
133    // P2SEL1 |= BIT0;
134    // P2SEL0 |= BIT0;
135     P1DIR &= ~BIT6; //configure P1.6 as input since it is connected to the compo
136
137
138     P2SEL1 |= (BIT0 | BIT1); // USCI_A0 UART operation
139     P2SEL0 &= ~(BIT0 | BIT1);
140
141     CSCTL0_H = CSKEY_H;                     // Unlock CS registers
142     //CSCTL1 = DCOFSEL_3 | DCORSEL;           // Set DCO to 8MHz
143     CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
144     //CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;   // Set all dividers
145     CSCTL0_H = 0;                           // Lock CS registers
146
147     // Configure USCI_A0 for UART mode
148     UCA0CTLW0 = UCSWRST;                    // Put eUSCI in reset
149     UCA0CTLW0 |= UCSSEL__SMCLK;             // CLK = SMCLK
150     // Baud Rate calculation
151     // 8000000/(16*9600) = 52.083
152     // Fractional portion = 0.083
153     // User's Guide Table 21-4: UCBRS0 = 0x04  0x11
154     // UCBRF0 = int ( (52.083-52)*16) = 1
155     UCA0BRW =  104;        //52     104=16MHz/16/9600              // 8000000/16
156     UCA0MCTLW |= UCOS16 | UCBRF_2 | 0x4900;////UCBRF_1
157     UCA0CTLW0 &= ~UCSWRST;                  // Initialize eUSCI
158     UCA0IE |= UCRXIE;                       // Enable USCI_A0 RX interrupt
159
160     __delay_cycles(6000);
161
162     // Configure TA2 as counter with COUT as input clock, enable overflow inter
163     TA2CTL = TASSEL__TACLK | MC__CONTINUOUS | TACLR | TAIE;
164     // Configure TB0 as counter with TB0CLK/P2.0 as input clock, enable overflo
165     //THIS IS USED IF EXTERNAL COMPARATOR IS USED
166     //TB0CTL = TASSEL__0 | MC__CONTINUOUS | TBCLR | TBIE;
167
168     // Configure COMP and P1.2 as C2 input
169     CECTL0 = CEIPEN | CEIPSEL_2;
170     CECTL1 = CEPWRMD_1;
171     CECTL2 = CEREFL_2 | CERS_3 | CERSEL;
172     CECTL3 = BIT2;
173     CECTL1 |= CEON;
174
175     // Configure TA1 timer to provide interval
176     TA1CCR0 = (SYSTEM_SMCLK / SAMPLE_FREQUENCY) - 1;
177     TA1CCR1 = (SYSTEM_SMCLK / SAMPLE_FREQUENCY / 2) - 1;
178     TA1CCTL1 = OUTMOD_3;
```

```
179     TA1CTL = TASSEL__SMCLK | ID__1;
180
181     // Configure ADC12 for sampling
182     ADC12CTL0 = ADC12SHT0_0 | ADC12SHT1_0 | ADC12ON;
183     ADC12CTL1 = ADC12SHS_4 | ADC12CONSEQ_2 | ADC12SHP;
184     ADC12CTL2 = ADC12RES_2 | ADC12DF_0; //ADC12DF_0: Unsigned 12-bit read, ADC12
185     ADC12MCTL0 |= ADC12INCH_0;
186     ADC12CTL0 |= ADC12ENC | ADC12SC;
187
188     // Configure DMA channel 0 to transfer ADC samples
189     __data20_write_long((uintptr_t)&DMA0SZ, (uint16_t)SAMPLE_LENGTH); //
190     __data20_write_long((uintptr_t)&DMA0SA, (uintptr_t)&ADC12MEM0);
191     __data20_write_long((uintptr_t)&DMA0DA, (uintptr_t)&circularBuffer[0]);
192     DMACTL0 = DMA0TSEL__ADC12IFG;
193     DMA0CTL = DMADT_0 | DMASRCINCR_0 | DMADSTINCR_3 | DMAIE | DMAEN;
194
195     // Initialize DMA buffer pointers
196     previousBuffer =&circularBuffer[SAMPLE_LENGTH]; ////////////////////&circula
197     activeBuffer = &circularBuffer[0];
198     nextBuffer = previousBuffer;
199
200     //Initialize FFT variables
201     uint16_t shift;
202     msp_fft_q15_params fftParams;
203     msp_cmplx_fft_q15_params fftParams_cmplx;
204     msp_abs_q15_params absParams;
205     msp_cmplx_conj_q15_params cmplxParams;
206     cmplxParams.length=1024;
207     absParams.length=1024;
208     tmpParams.length= 6;
209
210     //msp_cmplx_fft_q15_params fftParams;
211
212     /* Initialize the fft parameter structure. */
213     fftParams.length = 1024; ////////////////////////IT WAS 128 //////////////
214     fftParams.bitReverse = 1;
215     fftParams.twiddleTable = msp_cmplx_twiddle_table_1024_q15; ////////////////
216     // Initialize FIR pointers
217     FilterIndex = 2*FIR_LENGTH-COEFF_LENGTH;
218
219     /* Initialize the FIR parameter structure. */
220     firParams.length = FIR_LENGTH;
221     firParams.tapLength = COEFF_LENGTH;
222     firParams.coeffs = FILTER_COEFFS_EX1;
223     firParams.enableCircularBuffer = true;
224
225     /* Zero initialize FIR input for use with circular buffer. */
226     fillParams.length = 2*FIR_LENGTH;
227     fillParams.value = 0;
228     status = msp_fill_q15(&fillParams, circularBuffer);
229     msp_checkStatus(status);
230
231     /*Max Value function initialize */
232     firmaxParams.length = FIR_LENGTH;
233     firminParams.length = FIR_LENGTH;
234
235     _q15 q15MaxVector;
236     uint16_t uint16MaxIndex;
237     uint16MaxIndex=0;
238
```

```
239
240     //Initialize general variables
241     delta_previous=0;
242     delta=0;
243     spikes_avoid=0;
244     redundancy=0;
245     TX_Buffer=0;
246     ppp=0;
247     kk=0;
248     ZERO_C=0;
249     average=0;
250     i=0;
251     IDLE=1;
252     IDLE_TIME=0;
253     IDLE_TIME_MAX=256;
254     temp=0;
255     caption=0;
256     DataCopyPointer=0;
257
258     TA1CTL |= MC__UP | TACLR;
259
260     while(1) {
261         cycle_counts = 0;
262         temp2=temp;
263         if (caption==1)
264             {
265             caption=0;
266             memcpy(&result[0], circularBuffer, FIR_LENGTH * sizeof(_q15) );
267             memcpy(&result[FIR_LENGTH], result, FIR_LENGTH * sizeof(_q15) );
268             memcpy(&result[2*FIR_LENGTH], result, 2*FIR_LENGTH * sizeof(_q15) );
269
270             status = msp_fft_auto_q15(&fftParams, result, &shift);
271             msp_checkStatus(status);
272
273             index_imag=0;
274             index_real=0;
275
276             for(kk=1;kk<512;kk++){
277                 if(kk%2==0){//IMAGINARY
278                     result[512+index_imag]=result[kk];
279                     index_imag++;
280                 }
281                 else{  //REAL
282                     result[index_real]=result[kk];
283                     index_real++;
284                 }
285             }
286
287
288             for(kk=0;kk<512;kk++){
289                 result[kk]= sqrt((pow(result[kk],2))+(pow(result[kk+512],2)));
290             }
291
292             msp_checkStatus(status);
293
294             memcpy(spectrum_value, &result[2], 4* sizeof(_q15) );
295
296             status = msp_max_q15(&tmpParams,spectrum_value ,&q15MaxVector,&fundam
297             msp_checkStatus(status);
298
```

```
299             fundamental_index=fundamental_index+3;
300
301             Data_To_Transmit[0]= q15MaxVector;
302
303             for(kk=1;kk<16;kk++){
304                 memcpy(spectrum_value, &result[(kk+1)*fundamental_index-3], 6* si
305
306                 status = msp_max_q15(&tmpParams,spectrum_value ,&q15MaxVector,&ui
307                 msp_checkStatus(status);
308
309                 Data_To_Transmit[kk]=q15MaxVector;
310
311             }
312
313             UCA0TXBUF='k';
314             __delay_cycles(800000);
315
316           for(kk=0;kk<16;kk++){
317              for(ppp=0;ppp<2;ppp++){
318
319                  if(ppp==1){
320                      TX_Buffer = (uint8_t)(Data_To_Transmit[kk] >> 8); //MSBs
321
322                  }
323                  else if(ppp==0){
324                      TX_Buffer = (uint8_t)(Data_To_Transmit[kk]& 0x00FF); //LSBs
325
326                  }
327
328                       UCA0TXBUF =TX_Buffer;
329                       __delay_cycles(800000);
330                 }
331
332             }
333
334          /* for(kk=0;kk<200;kk++){  //PADDING FOR MISSING CHAR AT RX
335                UCA0TXBUF =0x0D;
336                __delay_cycles(80000);
337           }*/
338
339             P4OUT^=BIT4;
340             }
341
342
343
344
345     while(cycle_counts<8*8*4)
346     {
347         // Enter LPM0 and wait for DMA ISR
348         __bis_SR_register(LPM0);
349
350
351         P4OUT ^= BIT5;
352
353         ///AVOID DETECTION AT THE END
354         if(IDLE==1){
355             IDLE_TIME++;
356             if(IDLE_TIME >= IDLE_TIME_MAX){
357                 IDLE=0;
358                 IDLE_TIME=0;
```

```
359             }
360         }
361
362
363
364         // Run FIR filter
365         status = msp_fir_q15(&firParams, &circularBuffer[FilterIndex], &result[sa
366         msp_checkStatus(status);
367         FilterIndex ^= FIR_LENGTH;
368
369
370
371
372         if((cycle_counts+1)%4==0){
373             status = msp_max_q15(&firmaxParams,result ,&temp,&uint16MaxIndex);
374             msp_checkStatus(status);
375             status = msp_min_q15(&firminParams,result ,&temp2,&uint16MaxIndex);
376             msp_checkStatus(status);
377             delta_previous=delta;
378             delta=temp-temp2;
379         }
380
381
382 //Compare max with previous (with temp and temp2 we avoid to detect as event th
383
384         if (  (delta-delta_previous>20)   && caption==0 && IDLE==0 && delta_previ
385             caption+=1;
386             DataCopyPointer++;
387             IDLE=1;
388
389             P4OUT ^= BIT4;  //LED
390         }
391
392         else { ///IT WAS ONLY IF
393             temp=q15MaxVector;
394         }
395
396         cycle_counts++;
397         // DEBUG: Toggle P4.5
398         P4OUT ^= BIT5;
399
400
401         }
402         //P4OUT ^= BIT4;
403         i+=1;
404 }
405 }
406     // DMA interrupt service routine
407 #pragma vector = DMA_VECTOR
408 __interrupt void DmaIsr(void)
409 {
410     switch(__even_in_range(DMAIV, DMAIV_DMA0IFG)) {
411     case DMAIV_NONE: break;
412     case DMAIV_DMA0IFG:
413         // Setup next transfer
414         __data20_write_long((uintptr_t)&DMA0DA, (uintptr_t)nextBuffer);
415         DMA0CTL |= DMAEN;
416
417         // Switch buffer pointers
418         previousBuffer = activeBuffer;
```

```
        activeBuffer = nextBuffer;
        nextBuffer = previousBuffer;

        // Exit from LPM0
        __bic_SR_register_on_exit(LPM0_bits);
        break;
    default: break;
    }
}
```

10.2.1 Generate Neural Network Training Dataset

```
%ALLINEO SEGNALI E CALCOLO VARIANZA MEDIA PER CREARE UN RUMORE ADATTO

index=1
for p=1:length(names_unique) %%1 to 113
    p
    start= (p-1)*10+1;
    ending= start+9;
 for k=1:200
        for z=1:14

          test_harmonics(z,index)=harmonics_composed(z,start+ceil(rand(1,1)*6))+
          sqrt(var(harmonics_composed(z,start:ending)'))*randn(1,1);

        end
        index=index+1;
 end
end



training_result=zeros(length(names_unique),200*length(names_unique));

index=1;
    for p=1:length(names_unique)
        p
        for k=1:200
            training_result(p,index)=1;
            index=index+1;

        end
    end

    training_result_VERIF=zeros(length(names_unique),3*length(names_unique));


index=1;
    for p=1:length(names_unique)
        p
        for k= 1:3


         TEST_WAVES(:,index)= harmonics_composed(:,(p-1)*10+k+7);
            training_result_VERIF(p,index)=1;
```

```
            index=index+1;

        end
    end
```

10.2.2 Read Signal from COM and Classify it

```
clear UART_wave
clear UART_wave_unique

clear start
warning('off','all')


appliance_measurement=1; %%STATE WHAT IS THE CORRESPONDING APPLIANCE

if(not( exist('device')))
device = serialport("COM7",9600);
end
flush(device)
flush(device,'input')
flush(device,'output')


start= read(device,1,"uint8");

while(isempty(start))
    start= read(device,1,"uint8");
end
disp("TRANSMISSION DETECTED")
UART_wave= read(device,3*512,"uint8");
%%UART_wave= read(device,256,"uint16") %%%%%%%%%%%%


%%
unique_index=1;
k=3;
while(k<=length(UART_wave))
     value_1= UART_wave(k-2);
     value_2= UART_wave(k-1);
     value_3= UART_wave(k);

     eq12=0;
     eq13=0;
     eq23=0;

     if(value_1 == value_2)
         eq12=1;
     end
     if (value_1==value_3)
         eq13=1;
     end
     if (value_2==value_3)
         eq23=1;
     end
```

```
49
50
51      if(  eq12==1)
52          new_value=value_1;
53
54          if(eq23==1)
55              k= k+3;
56          elseif (eq23==0)
57              if(value_3 == UART_wave(k+1) | value_3 == UART_wave(k+2))
58                  k=k+2;
59
60              else
61              k=k+3;
62              end
63
64          end
65
66      elseif(eq13==1)
67          new_value=value_1;
68          k=k+3;
69      elseif(eq23==1)
70          if(value_1 == UART_wave(k-3) | value_3 == UART_wave(k-4))
71              k=k+2;
72          else
73              k=k+3;
74          end
75          new_value=value_2;
76      else
77          k=k+3;
78      end
79
80
81      UART_wave_unique(unique_index)=new_value;
82      unique_index=unique_index+1;
83
84  end
85
86  if(UART_wave_unique(1) > UART_wave_unique(2))
87      start_index=1;
88  else
89      start_index=2;
90  end
91
92  clear data;
93  if(rem(length(UART_wave_unique)-start_index+1,2)==1)
94      data=UART_wave_unique(start_index: end-1);
95  else
96      data= UART_wave_unique(start_index: end);
97  end
98
99  index=1;
100 % if(UART_wave(1)>5*UART_wave(2)
101 for i=2:2:length(data)
102     signal(index) =bitshift(data(i),8)+  data(i-1);
103     if(signal(index)>6000)
104         index
105         i
106     end
107     index=index+1;
108
109 end
```

```

repeated_wave= cat(2,signal,signal);
repeated_wave= cat(2,repeated_wave,repeated_wave);


plot(repeated_wave)
CLASSIFY(appliance_measurement,repeated_wave');
```

10.2.3 MQTT - Subscribe and Read Messages from predefined Topics

```
clear channel_1
clear channel_2
clear my_Broker
clear num_messages1
clear num_messages2
clear message1
clear message2
clear data_rx1
clear data_rx2
clear data_rx


my_Broker=mqtt('tcp://192.168.0.102')

channel_1 = subscribe(my_Broker, 'Broker/To/cc32xx','QoS', 2);
channel_2 = subscribe(my_Broker, 'cc32xx/ToggleLED1','QoS', 2);

num_messages1= channel_1.MessageCount;
num_messages2= channel_2.MessageCount;
message1=0;
my_index=1;

while(1)
    count1=channel_1.MessageCount;
    count2=channel_2.MessageCount;
   if( count1 >  num_messages1)
       message1=read(channel_1)
       num_messages1= channel_1.MessageCount;
   end
   if( count2 >  num_messages2)
       message2=read(channel_2)
       my_index=my_index+1;
       num_messages2= channel_2.MessageCount;

   end
   pause(0.1);
end

data_rx1=double(cell2mat(num2cell(message1{:})));
data_rx2=double(cell2mat(num2cell(message2{:})));

data_rx(1:256)=data_rx1;
data_rx(257:512)=data_rx2;

```

```
dataconversion=double(table2array(dataconversionS2))

for i=length(data_rx):-1:1
    for k=1:27
   if(data_rx(i)== dataconversion(k,2))
            data_rx(i)= dataconversion(k,1);

   end
    end
end

for i=length(data_rx):-1:1
    if(data_rx(i) == 65533)
        if((data_rx(i-2)~=65533 )&& (data_rx(i+2)~=65533 ))
            data_rx(i)=(data_rx(i+2)+data_rx(i-2))/2
        else
            data_rx(i)=(data_rx(i+4)+data_rx(i-4))/2
        end
    end
end


index_rx=1;
for i=1:2:length(data_rx)
    SIGNAL_RX(index_rx)= data_rx(i)+ bitshift(data_rx(i+1),8);
    index_rx=index_rx+1;
end
plot(SIGNAL_RX);
```

10.2.4 Automatic Waveforms acquisition from MQTT

```
%% Read all Appliance Names
% clear
list_appliances= dir("Appliances");

for i=3:length(list_appliances)
    appliances_names(i-2)= string(strcat("./Appliances/",string(list_appliances(i
    names(i-2)= string(extractBefore(list_appliances(i).name, strlength(list_appl
end

for i=3 : length(list_appliances)
    TMP=string(extractBefore(list_appliances(i).name, strlength(list_appliances(i
    MK_value=string(extractAfter(TMP, strlength(TMP)-3));
    if(strcmp(MK_value,"MK1")==1)
        correction_factor(i-2) =61.4835;
    elseif(strcmp(MK_value,"MK2")==1)
        correction_factor(i-2) =60.200;
        % end
    elseif(strcmp(MK_value,"MK3")==1)
        correction_factor(i-2) =60.9562;
        % end
    end

end
```

```
24
25 names_unique= unique(names,'stable');
26 number_samples=cell2mat(cellfun(@(x) sum(ismember(names,x)),names_unique,'un',0))
27
28 %% CREATE 1050 ELONGED AUDIO FILES
29
30  for p=1:1050
31     p
32      [data,fs]=audioread(appliances_names(p));
33      data(:,2)=4*data(:,2);
34      START_COPY=1;
35      END_COPY=882;
36      aaa=cat(1,data(START_COPY:END_COPY,2),data(START_COPY:END_COPY,2));
37      aaa=cat(1,cat(1,aaa,aaa),cat(1,aaa,aaa));
38      aaa=cat(1,cat(1,aaa,aaa),cat(1,aaa,aaa));
39      aaa=cat(1,cat(1,aaa,aaa),cat(1,aaa,aaa));
40      aaa=cat(1,cat(1,aaa,aaa),cat(1,aaa,aaa));
41      IRes=cat(1,aaa(287000:end),data(END_COPY+1:end,2));
42      filename=sprintf("./Appliances_Sounds/Appliance_\%d.wav", p);
43      audiowrite(filename,IRes,44100);
44  end
45      plot(IRes);
46
47 %% AUTOMATIC PLAY OF THE AUDIO FILES AND MEASUREMENT FROM THE WIFI
48
49 %%SELECT INTERVAL OF APPLIANCE SOUNDS TO PLAY:
50 start=41;
51 stop=50;
52
53 my_Broker=mqtt('tcp://mqtt.eclipse.org')
54 channel_1 = subscribe(my_Broker, 'Broker/To/cc32xx','QoS', 2);
55 channel_2 = subscribe(my_Broker, 'cc32xx/ToggleLED1','QoS', 2);
56
57 num_messages1= channel_1.MessageCount;
58 num_messages2= channel_2.MessageCount;
59 message1=0;
60 message2=0;
61
62 read_channel_1=0;
63 read_channel_2=0;
64 p=start;
65 load('data_conversion.mat');
66 dataconversion=double(table2array(dataconversion));
67
68 while  (p<=stop)
69     close all;
70     p
71     play_filename= sprintf("./Appliances_Sounds/Appliance_%d.wav", p);
72     [sound_array,Fs]= audioread(play_filename);
73     sound= audioplayer(sound_array, Fs);
74     play(sound)
75     read_channel_1=0;
76     read_channel_2=0;
77     watchdog=0;
78     %%WAIT FOR BOTH TOPICS TO RECEIVE A MESSAGE
79     while( ~(read_channel_1 && read_channel_2))
80         count1=channel_1.MessageCount;
81         count2=channel_2.MessageCount;
82
83         if( count1 >  num_messages1)
```

```
84              message1=read(channel_1)
85              num_messages1= channel_1.MessageCount;
86              read_channel_1=1;
87  %             watchdog=0;
88          end
89
90          if( count2 >  num_messages2)
91              message2=read(channel_2)
92              num_messages2= channel_2.MessageCount;
93              read_channel_2=1;
94  %             watchdog=0;
95          end
96
97          pause(0.1);
98      end
99
100
101     data_rx1=double(cell2mat(num2cell(message1{:})));
102     data_rx2=double(cell2mat(num2cell(message2{:})));
103
104     data_rx(1:256)=data_rx1;
105     data_rx(257:512)=data_rx2;
106
107
108     for i=length(data_rx):-1:1
109         for k=1:27
110             if(data_rx(i)== dataconversion(k,2))
111                 data_rx(i)= dataconversion(k,1);
112             end
113         end
114     end
115
116     for i=length(data_rx):-1:1
117         if(data_rx(i) == 65533)
118             if((data_rx(i-2)~=65533 )&& (data_rx(i+2)~=65533 ))
119                 data_rx(i)=(data_rx(i+2)+data_rx(i-2))/2
120             else
121                 data_rx(i)=(data_rx(i+4)+data_rx(i-4))/2
122             end
123         end
124     end
125
126
127     index_rx=1;
128     for i=1:2:length(data_rx)
129         SIGNAL_RX_appliance3(p,index_rx)= data_rx(i)+ bitshift(data_rx(i+1),8);
130         index_rx=index_rx+1;
131     end
132     plot(SIGNAL_RX_appliance3(p,:));
133
134     prompt='Is the Wave camptured ok? [y/n] ------>  ';
135     user_input= input(prompt,'s')
136
137     if(strcmp(user_input,'y')==1)
138         p=p+1;
139         disp('Next Wave!');
140     else
141         disp('Repeating current Wave!');
142     end
143 end
```

References

[1] A. Torelli (a.y. 2019-2020). Master's Thesis: *Non-Intrusive Load Monitoring in Smart Home Application*. Polytechnic of Milan

[2] E.J. Aladesanmi, KA Folly (2015). *Overview of non-intrusive load monitoring and identification techniques*. IFAC (International Federation of Automatic Control)

[3] R.Gopinatha, MukeshKumarab, C.Prakash Chandra Joshuaa, KotaSrinivasab (Nov. 2020. *Energy management using non-intrusive load monitoring techniques – State-of-the-art and future research directions*. Sustainable Cities and Society (SCS)

[4] Klemenjak, C., Goldsborough, P. (2016). *Non-intrusive load monitor-ing: A review and outlook.* arXiv preprint arXiv:1610.01191

[5] Liu, H. (2020). *Non-Intrusive load monitoring: Theory, Technologiesand Applications. Springer Verlag, Singapore.*

[6] R.S., and Semwal, S. (2013). *A Simplified New Procedure for Identification of Appliances in Smart Applications.* 2013 IEEE International System Conference (SysCon2013) Proceedings, pp.339-344.

[7] Nalmpantis, C., Vrakas, D. (2019). *Machine learning approaches fornon-intrusive load monitoring: from qualitative to quantitative compa-ration.* Artificial Intelligence Review, 52(1), 217-243.

[8] Hamdi, M., Messaoud, H., Bouguila, N. (2020). *A new approach ofelectrical appliance identification in residential buildings.* Electric PowerSystems Research, 178, 106037.

[9] Zheng, Z., Chen, H., Luo, X. (2018). *A supervised event-based non-intrusive load monitoring for non-linear appliances.* Sustainability, 10(4),1001.

[10] Bonfigli, R., Squartini, S., Fagiani, M., Piazza, F. (2015, June). *Un-supervised algorithms for non-intrusive load monitoring: An up-to-dateoverview.* In 2015 IEEE 15th International Conference on Environmentand Electrical Engineering (EEEIC), 1175-1180.

[11] F. Luo, G. Ranzi, W. Kong, Z. Y. Dong, S. Wang, and J. Zhao (2017) *Non-intrusive energy saving appliance recommender system for smart grid residential users.* IET Gener. Transm. Distrib., vol. 11, no. 7, pp. 1786–1793.

[12] M. Kahl, A. Haq, and T. Kriechbaumer (2016) *Whited-a worldwide household and industry transient energy data set.* 3rd International Workshop on Non-Intrusive Load Monitoring.

[13] T. Liu, Y. Liu, Y. Che, S. Chen, Z. Xu, and Y. Duan (2014). *SHE: Smart home energy management system for appliance identification and personalized scheduling.* Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication, 2014, pp. 247–250.

[14] Mayhorn, E. T., Sullivan, G. P., Petersen, J. M., Butner, R. S., Johnson, E. M. (2016). *Load disaggregation technologies: real worldand laboratory performance.* Pacific NorthwestNational Lab.(PNNL), Richland, WA (United States).

[15] W. Martin (2 Oct. 2019). *A Review of Non-Intrusive Load Monitoring Tracking Approaches.* https://medium.com/@VervEnergy/a-review-of-non-intrusive-load-monitoring-tracking-approaches-9b3b3536143f