Power Quality Monitoring and Data Collection

High frequency analysis of line voltage

Thomas de Graaf

***Abstract* - The increased use of power electronics devices and more decentralized power generation has attributed to more issues regarding power quality. To study the sources of these issues a cheap power quality monitor was designed that can measure some power quality parameters with an interval of 2 seconds. This paper shows how to expand its functionality to do line voltage analysis on a higher frequency, while adding the least amount of components possible. The usability of the analog to digital converter (ADC) of the microcontroller (ESP32) present on the power quality monitor is assessed. It is determined that a sampling frequency of 500 kHz is needed for troubleshooting a voltage grid [1] This is sufficient to measure many harmonics of the 50 Hz grid and some impulsive phenomena. However, it might not be enough for showing phenomena caused by fast switching of switched-mode power supplies etc. Using the internal ADC a maximum sampling frequency of 768 kHz is achieved using I2S, which meets the requirements. The signal conditioning to make the line voltage measurable is however not sufficient for the ADC as voltage levels are too low. Because of this a 10 kHz test signal is considered as well as two static voltages. The FFT magnitude plot and performance of static voltage readings are compared for different sampling frequencies. It is found that noise performance increases with higher sampling frequencies. An alternative signal conditioning system that meets the requirements is proposed.**

1 Introduction

In the modern world, Power Quality (PQ) monitoring is becoming more and more important. This is partly due to the increased usage of Power Electronics (PE) devices and non-linear loads, but also because we are moving towards grids with decentralised power generation [2, 3]. The move towards decentralised generation (and/or microgrids) in combination with the high usage of PE devices makes it harder to assess power quality from a limited number of measurement nodes as is often the case today.

More detailed information within the grid is needed to find PQ problems and their sources to ensure that the PQ parameters are within standards for every user.

To move towards this goal, a cheap PQ monitor that that can capture the basic PQ parameters with an interval of 2 seconds has been created in [2]. This paper focuses on expanding this device to allow for voltage analysis with a sampling rate above 500 kHz while keeping costs low. It would be beneficial to add the least amount of components as possible.

High frequency analysis could reveal the sources of PQ issues that cannot be indicated with long measurement intervals. An example could be fig. 1 below. As can be seen there is a dip in the voltage that can only be seen with a higher sampling rate.

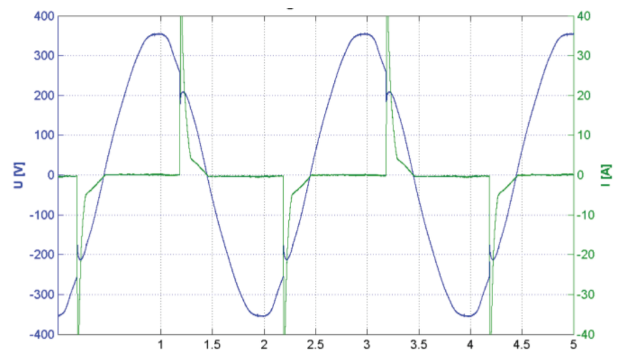

Figure 1: An example of a disturbed grid voltage [4].

2 Problem and research question

To make high frequency PQ monitoring on a large scale possible, the costs of the measurement devices and (data) infrastructure should be kept low. However, the low cost should not lead to unusable data. The functionality of the PQ monitor created in [2] will be expanded to allow for high frequency line voltage analysis.

This could tell us more about PQ/EMI related failures. Examples of high frequency phenomena are lightning strikes and power electronics related phenomena, e.g. switching frequencies. Therefore this paper will try to answer the research question: *What frequencies are important for PQ/EMI analysis and can they be measured with the limited hardware available?*

3 Related work

To answer the research question stated above, it would be beneficial to start with a definition of PQ, see what frequencies are of interest and what the capabilities of the existing PQ/EMI monitor are.

3.1 Power Quality (PQ) basics

PQ parameters are a subset of Electromagnetic Compatibility (EMC) and can tell us about the extend to which a power grid is stable [2]. Problems with PQ can lead to the failure of an entire system, like the emergency backup power of a hospital not working due to large inrush current of power electronics converters [3]. This could in extreme cases also happen to the national power grids, making it very important to have insights in the PQ on a grid and mitigate issues wherever possible. In the news you can already hear a lot about grid operators not allowing new connections in the Netherlands out of concern for PQ issues [5, 6]. Grids have predetermined operating conditions and maxima for several electrical parameters, which can be used for this mitigation [2]. The extend to which these parameters adhere to the standards determine the quality of an electrical network. Poor PQ can be very bad for electrical devices [2].
Over the past years, there has been an increase in the amount of PQ monitoring around the world [1]. Monitoring gives network operators or utilities more information about the performance of their network, allowing them to take action against PQ problems [1]. Measuring PQ is getting more and more important due to several factors. Examples are the move towards more distributed electricity generation (e.g. solar panels and wind turbines), more non-linear loads (e.g. power converters) and electric vehicles [2, 3, 1]. These devices lead to a less stable environment regarding PQ compared to more traditional resistive loads (e.g. incandescent lamps or heaters).

3.1.1 PQ parameters

Now that it is know what the basic meaning of PQ and its importance are, some parameters that make up power quality should be discussed. Although a lot of network operators are installing PQ monitoring devices, there is a lack of knowledge and agreement on the location, amount and parameters to measure [1]. According to CIGRE/CIRED, the following parameters can be measured (depending on the monitoring objective) [1]:

- (Steady state) RMS voltage;
- (RMS) current and harmonic currents;
- Voltage unbalance, harmonics, THD or flicker;
- Voltage sags, swells and rapid changes;
- High/low RMS voltage per 1-10 min window;
- Transients;
- Frequency, P, Q, power factor, phase shift.

Analyzing the voltage at a high frequency would give more insights regarding transients, frequency contents (harmonics, etc.) and rapid changes. As most PQ parameters are based on the line voltage, it is chosen not to look at the current for this project.

3.2 Frequencies of interest

For this research, it is important to know what bandwidth is important to capture most PQ parameters. It would be interesting however to see just how high the bandwidth can get with the limited hardware that is provided. In the future, this information could then be used to see if it is feasible to even capture EMI phenomena (which can also be a part of PQ).
The main frequency we would like to measure in the Dutch power grid is the 50 Hz signal, its harmonics and change over time. Suppose we would like to measure up to the 20th harmonic, we would need a bandwidth of just 1 KHz. The quality and disturbances of the 50 Hz signal is the most important measurement in PQ analyses and because of the low bandwidth relatively easy to measure. It will become trickier when EMI (e.g. switched mode power supplies) and impulsive phenomena are taken into account. In the book [7] it can be found that impulsive PQ phenomena like lightning strikes last only a few microseconds. Let's suppose a lightning strike lasts 3 microseconds, then a bandwidth of 333.33 kHz would be needed to capture the event. This in turn means a minimum sampling rate of 666.67 kHz, or 0.667 MHz, to meet the Nyquist criterion.
Another example is the increased use of switched mode power supplies. From the same book, page 104-105, it can be seen that DC/DC converters often use a high switching frequency in the kilohertz range. The example on page 105 has a switching frequency of 500 kHz, or a Nyquist frequency of 1 MHz just to see the fundamental frequency of the Power Supply Unit (PSU) [7]. According to CIGRE/CIRED JWG C4.112 a sampling rate of 500 Khz will suffice for most (troubleshooting) applications (table I, monitoring objective "Troubleshooting") [1]. From this information it can be concluded that if we would like to show information regarding lightning, EMI and other higher frequency PQ parameters the sampling rate would need to be at least 500 kHz, preferably in the MHz range.

3.3 Measuring PQ

How to measure PQ depends on the goal and location of the measurement. Measurements can be done e.g. to do compliance verification, performance analysis, troubleshooting, etc. This can be done at Extra High Voltage (EHV), High Voltage (HV), Medium Voltage (MV) and Low Voltage (LV) networks, where in general the cost is highest for the EHV networks and the lowest for the Low Voltage networks [1].
Today, PQ is often measured at 2 main points. One at the grid operator level, for example at a distribution transformer, and another measurement is done at the point of common coupling (POCC) like the energy meter in one's home [2]. To study EMI in more detail, it would be beneficial to have more measurement locations at the electrical appliance level. This information can then be used to investigate the origin of the PQ problems and possibly mitigate them [2].

3.4 Cheap PQ monitoring [2]

To troubleshoot PQ at many nodes in a network, a cheap measurement device is needed. As discussed before it is cheapest to measure in a LV network like the 230 V network in Dutch houses.

To do this, specialized high-end equipment (e.g. MPQ1000 or Fluke 1770) with a lot of flexibility can be used. This allows the user to extract measured data and interpret it however they want. This includes raw line voltage, raw line current and calculated PQ parameters. However, the price of these units is often high [2]. Cheaper monitors are available from brands like TP-Link, but their flexibility is limited and the accuracy is often unknown [2]. In this project an inexpensive PQ monitor based on an ESP32 module is studied. In combination with an energy meter IC, resistive divider and shunt this monitor can achieve an energy measurement accuracy of less than 1%, comparable to a class B energy meter (EN-50470). This is done using the ATM90E26 from Microchip using the SPI bus of the ESP32. The ATM90E26 has onboard DSP to determine the PQ parameters, such that they do not have to be calculated using the ESP32. Parameters are fetched every two seconds [2]. Signals are also fed directly to the 12 bit successive-approximation-register (SAR) ADCs of the ESP32 via signal conditioning to allow for even more flexibility [2, 8]. These signals will be used for this project, specifically the line voltage. To visualize this system the block diagram can be seen in Fig. 3. As an extra, the module is fitted with an expansion header to allow for comparison of PQ parameters with environmental conditions and/or add more PQ measurement options in the future. PQ parameters currently measured by the device can be found in table 1 [2]. Data is saved every 15 seconds to a micro-SD card and can be retrieved by removing the SD card or using the special Android app and Bluetooth. At the time of writing of [2], the total BOM was approximately €100. In bulk, this would probably be a lot lower. This device would therefore, with a few improvements (mainly in communication) be a viable option to measure a lot of nodes in a LV network. It can basically be seen as a higher frequency, advanced household energy meter like the one in a lot of Dutch fuse boxes ('meterkasten').

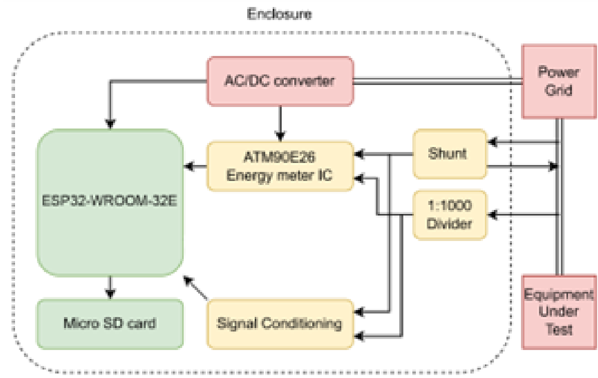

Figure 2: Block diagram of the PQ monitor [2].

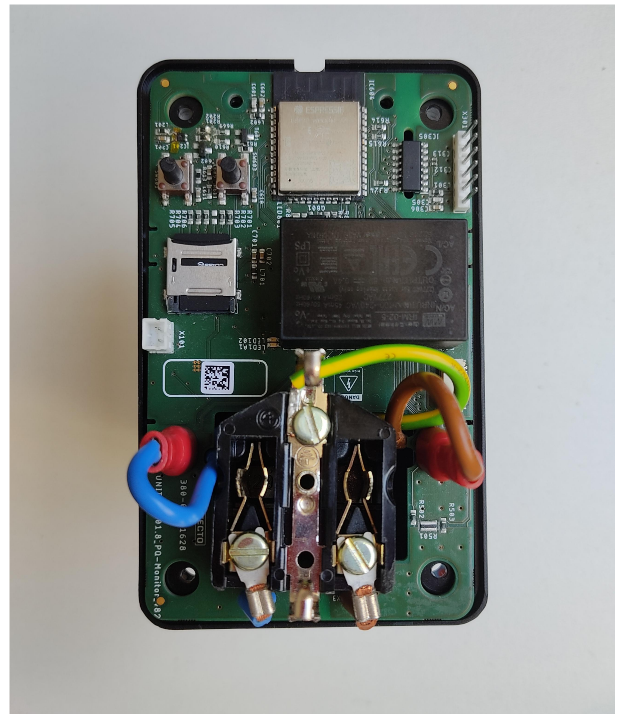

Figure 3: Photo of the PQ monitor.

Table 1: PQ parameters measured by the PQ monitor [2].

Elemental measurement	Power measurement	Energy measurements	Events
RMS voltage	Mean Active Power	Forward Active Energy	Sag
RMS current	Mean Reactive Power	Reverse Active Energy	Swell
Frequency	Mean Apparent Power	Absolute Active Energy	
Phase Angle		Forward Reactive Energy	
Power Factor		Reverse Reactive Energy	
		Absolute Reactive Energy	

The software is implemented using FreeRTOS, a real-time operating system which allows to create tasks and assign a priority to them. This means tasks are scheduled in order of importance [2]. This can reduce the amount of erroneous measurements and allow for easier expansion in the future.

4 Methodology

The research can be split into four main parts. These parts combined should answer second part of the research question. They can be summarized as follows:

1. Check ADC capabilities (get and verify data).
2. Check data for usability (data analysis).
3. Perform data analysis on the microcontroller.
4. Store data locally or on server.

The first part would consist of writing code for the ESP32 in the same framework as used in previous research. Important aspects are the maximum sampling rate at which we can still save data to RAM. The second part would be an analysis on frequency content, accuracy and precision. It would also be good to compare accuracy/precision with respect to sampling rate.
As a third step the analysis should be performed on the PQ monitor itself if that's possible. The last step would be to save the raw or analyzed data to the SD card present in the PQ monitoring device. If possible, it would be interesting to send data to a server. This would obviate the need to have physical access to the device.

4.1 Check ADC capabilities

To keep the costs of the PQ monitor low, research is done on the usability of the built-in ADC of the micro-controller (ESP32). If the built-in ADC is usable, probably no extra hardware is needed as signal conditioning measures are already in place [2]. Whether the signal conditioning is enough for the ESP32 has to be checked. Voltage readings are only linear between 100/150 mV and 2450 mV depending on the ADC attenuation setting [8]. With an oscilloscope, it can be checked whether the signal stays within these limits.
When this is verified, the maximum sampling rate can be determined. It has already been established that a sampling rate of 500 kHz would be a good minimum if we would like the frequency data for troubleshooting of PQ "errors". Different software implementations will be considered for the sampling process: direct sampling, sampling using Direct Memory Access (DMA) and I2S with DMA. When using DMA, the ADC has its own memory buffers to which it directly writes the samples. The main processor is not involved in this process. Using DMA in theory could lead to higher sampling rates as the processor only has to empty the memory buffers in time instead of constantly polling the ADC.
Data will be logged in its raw form, a 12 bit value, to the serial console.

4.2 Data analysis

To analyze the data from the serial console it will be imported into MATLAB. The accuracy and precision of voltage readings will be checked as well as the Fast Fourier Transform (FFT) of a "clean" 10 kHz sine wave as generated by an HP 33120A signal generator. From this it can be determined if the quality of the measured signals is sufficient. The quality of signals with a different sampling rate will be compared. Using this information it can be decided what sampling rate would be best to use. It preferably needs to be above 500 kHz as discussed before, but if the quality is low compared to lower sampling rates concessions might have to be made. Besides the FFT, interesting parameters could include minimum, maximum and mean voltage in a way narrower time span than is currently measured by the device. Data can be divided into timeslots of milli-/microseconds compared to 2 second slots as measured by [2].

4.3 Data storage

In the end, data should be stored on the SD card. This can be done by looking at existing code for the PQ monitoring device and reusing part of the code. Data could be saved as a csv file with a sample on each line and the time of the first sample in microseconds as the file name. The time would be with respect to the starting time of the device. However, if the actual time has been found using NTP, the timestamp would be in epoch time (the number of seconds since 1970). All samples are saved as a 12-bit value between 0 and 4095. This data could be imported in any software that supports a csv data stream.

5 Implementation, results and discussion

5.1 Get and verify data from an ADC

5.1.1 Direct sampling

From the three methods discussed above, sampling without DMA is ruled out. The official ESP32 documentation states that the theoretical maximum sampling rate would be 83.33 kHz [8]. On forums different results have been found ranging from 1 kHz to several kHz. One of the most promising results was using the adc1_get_raw function in a loop to achieve 27.1739 kHz [9].

5.1.2 DMA using only adc driver

To achieve a higher sampling rate the adc driver with DMA was implemented. In the end this was done by looking at the examples provided by the esp-idf and changing adc_dma_example to match the project requirements [10]. This meant changing the ADC input, removing unnecessary variable definitions regarding different versions of the ESP32 and changing the sampling rate to be above 500 kHz. Samples will be logged to the serial console after sampling. They are represented as a 12 bit value from 0 to 4095. The channel attenuation is set to 2.5 dB. The signal can be converted to a voltage using the formula [8]

$$V_{\text{meas}} = D_{\text{out}} \frac{V_{\text{max}}}{4095}, \tag{1}$$

where D_{out} is the measured value and V_{max} is 1.25 V. The resulting code can be found in Appendix A.
After changing the code, a 10 kHz sine wave from a NI MyDAQ is applied to the ADC input (10 kHz, 1 Vpp, 0.74 V DC-offset). The sampling rate is set to 1 MHz and data is logged to the console. Data is imported in MATLAB and converted to a voltage.
A plot can be found in Fig. 4, where the signal from the ADC is plotted against measurements using a Siglent SDS 1202X-E oscilloscope. Looking at the amount of samples per cycle revealed a perceived sampling rate of 166 kHz (16 samples per cycle). This is way lower than the expected 1 MHz. The sampling rate was decreased to 200 kHz, 83.33 kHz[1], 45 kHz and some values in between. The maximum perceived sampling rate is 166 kHz when setting the sampling rate in the code to 83.33 kHz. Any value above this yields a similar result. The maximum value of 83.33 kHz is in line with ESP32 documentation [8]. However, the sampling rate is way higher due to an error in esp-idf version 4.4 [11].

[1] At this point it was found that the expected range of the sampling rate (.sample_freq_hz) is 611 Hz - 83.33 kHz [8].

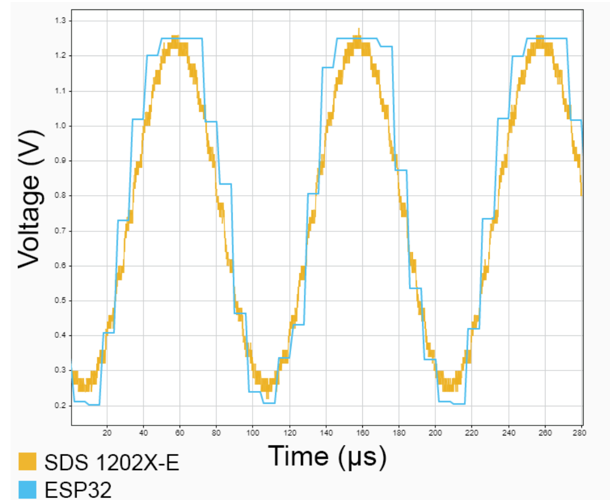

Figure 4: Plot of the measured signal using DMA with .sample_freq_hz = 1 MHz.

This is not fast enough for the intended use. Some further research revealed that the found result is the maximum speed using this method [8]. Further analysis on the samples generated using this method is therefore not done.
There should be another way to get closer to the theoretical maximum sampling rate of 2 MHz as described in the datasheet table 3-3 [12]. This turned out to be DMA using I2S.

5.1.3 DMA using I2S

After some research, the Inter-IC Sound Bus (I2S) seemed to be the most feasible way to get to high sampling rates. I2S was originally designed by Philips back in 1986 to allow digital audio devices to connect to each other. It is a bidirectional serial bus interface and mostly used for ADC/DAC applications [13]. Originally, I2S was probably designed with audio sampling rates in mind. In 1996 however, Philips published a new version of the I2S specification where a timing example is given for a data rate of 2.5 MHz [14]. This seems in line with the requirements of the project.
To get started with I2S I looked at the i2s_adc_dac peripherals example included in esp-idf 4.4.1. This code was simplified to just the sampling part and some code was created to represent the ADC values as integers from 0 to 4095. These represent a voltage from 0 to 3.3 volts. In the code most important parameters to set are:

- The physical ADC port;

- I2S mode;
- Sample rate and sample bits;
- Communication and channel format;
- Number of channels;
- Clock source;
- ADC port/channel.

For this project, the physical port is set to ADC1 channel 4, which corresponds to pin 8 of the ESP32 as shown in Fig. 5. This is the conditioned line voltage on the PQ monitor [15]. Note that all tests regarding measurement speed and stability are performed on an external ESP32 development module for flexibility and safety. Here, safety is both with regards to damaging the PQ monitor and physical safety (ground corresponds to the line voltage).
Besides this, sample bits is set to 16. Sample bits is set to 16 to get the 12 bit samples from the ADC while maintaining compatibility with the I2S driver. The communication format is set to the standard format as can be seen in Fig. 6. The complete code including other settings can be found in the code in Appendix B.

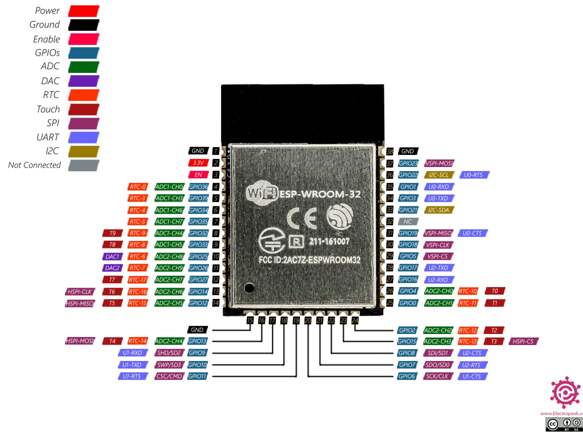

Figure 5: IO layout of the ESP32-WROOM-32E [16].

5.1.4 Handling I2S data

To start decoding the datastream, the sampling rate is simply set to 16 kHz and read into memory as type char*. The size of a char is 8 bits. Therefore, there are two chars per sample.
Samples can be decoded by looking at the communication format, which can be seen in Fig. 6. The first byte starts with the Most Significant Bit and the second data byte starts with the Least Significant Bit.

To represent the data as an unsigned integer, the second byte that starts with the LSB is converted

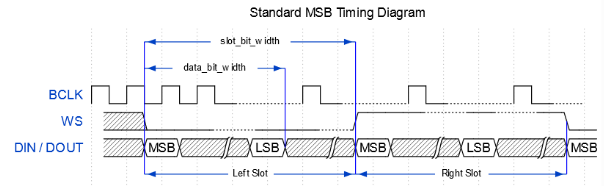

Figure 6: Data timing diagram [17].

to a 16 bit unsigned integer and shifted 8 bits left. Then a bitwise inclusive or is performed between this shifted data and the first byte. The resulting 16 bit value is padded, i.e. the first 8 bits are set to 0. The buffer is then printed to the serial monitor. C++ code for this can be found in line 95 - 108 of Appendix B. The data can now be saved by logging the serial monitor to a txt file.

5.1.5 Maximum frequency and results

Now that there is a way to get data using I2S from the ADC to the PC, the maximum sampling rate can be determined. The maximum sampling rate is the frequency before which measurements return implausible results or hardware crashes occur. The same 10 kHz signal from the MyDAQ is applied like before.
The sampling rate is increased in multiples of 48 kHz until results started to be unusable (in this case all zeroes). Increasing the sampling rate in multiples of 48 kHz seemed to work the best. This yielded the highest possible sampling rate of 768 kHz. The fact that this multiple works best might be due to the original application of I2S, which was in the domain of digital audio processing, where 48 kHz is a typical sampling rate [13].
Using I2S a sampling rate of 768 kHz could be reached[2]. This means the minimum sampling rate of 500 kHz as determined by CIGRE is reached [1].
Data is collected by using different sampling rates of 192, 384 and 768 kHz and applying a sine wave. The signal generator (HP 33120A) is set to 10 kHz with $V_{PP} = 1.000\ V$ and an offset of $700.0\ mV$. To get a better picture of the situation, a measurement with $F_s = 768\ kHz$ is shown in Fig. 7. It is superimposed on a measurement of the same signal using a Rohde und Schwarz RTB2002 oscilloscope. It can be seen that the signal is actually a 10 kHz sine with double the V_{PP} and an offset of $1400.0\ mV$, due to the output termination set to 50 Ω [18].

[2]Sampling rate is actually set to 384 kHz in code. There is a problem with the Hardware Abstraction Layer in esp-idf version 4.4 causing this issue [11].

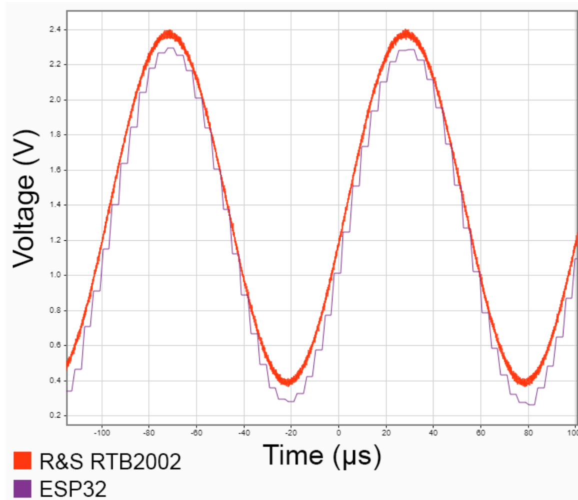

Figure 7: Plot of the measured signal using I2S with a sampling rate of 768 kHz.

As can be seen the plots mostly overlap except for a small voltage error. This can be explained by the accuracy of the ADC. In Fig. 8, the results of an experiment regarding linearity and offset is shown [19]. In this experiment, the ADC voltage is always a bit below the actual voltage (in the linear region). This corresponds to my findings.

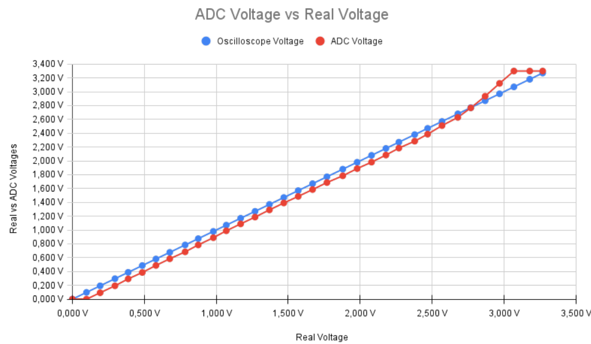

Figure 8: Input voltage versus output readings by the ADC [19]

5.1.6 Signal conditioning PQ monitor

To read the mains voltage using the ADC of the microcontroller, it is important to condition the signal. The signal should not damage the IC and should be readable. As we've already established in Sec. 4.1 the minimum voltage would be 150 mV w.r.t. ground. The maximum voltage would preferably be below 2450 mV to stay within the linear region of the ADC. Fig. 8 shows this in more detail. As the ground of the PQ monitor is floating on mains, a DSO138 clone is used to measure the conditioned line voltage. This is a small oscilloscope-like device with limited bandwidth, accuracy and precision. However, it can easily be used to display 50Hz signals. The great advantage is that it can be powered using a 9 V battery. This means it can operate completely isolated from mains.

Connecting the DSO138 and measuring on pin 8 of the ESP32 shows a voltage of 0 volts with respect to ground. This is verified by measuring with the ADC. From this it can be concluded that the signal conditioning is not sufficient. To properly troubleshoot a (micro)grid it would be beneficial to analyze the entire mains waveform. As nothing can be measured currently a new circuit has to be designed or the current one has to be modified.

In the current implementation the mains voltage is divided by 1000 and fed into IN+ of an LTC2055 op-amp. Fig. 10 shows how node N$21 is the neutral voltage divided by 1000. In Fig. 9 it can be seen that this voltage is being fed into the IC601.1. This op-amp is not connected to any negative supply and thus the output can only be half of the waveform. This is verified with a measurement shown in Fig. 11.

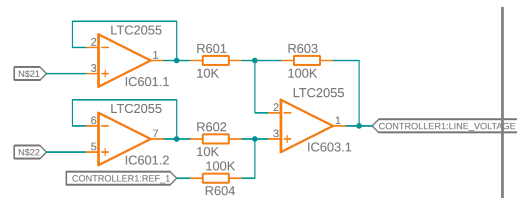

Figure 9: Signal conditioning measures as currently implemented [15].

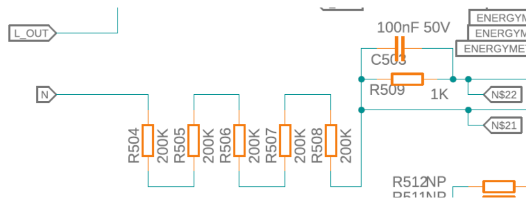

Figure 10: It can be seen that node 21 is 1/1000th of the line voltage [15]. (N22 is connected to phase as in [20])

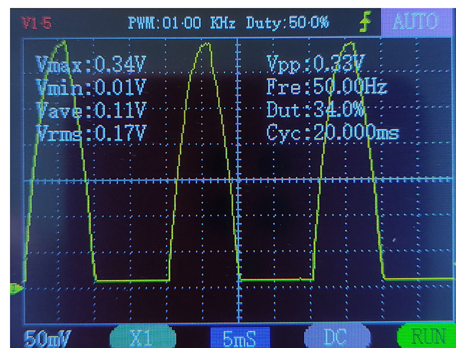

Figure 11: A measurement of the output of IC601.1.

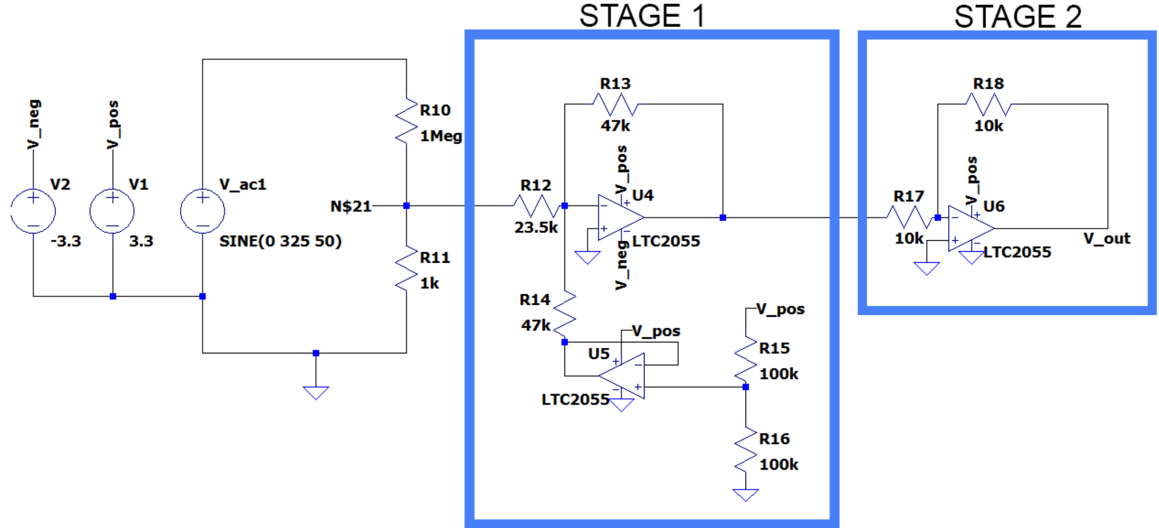

Figure 12: Schematic of possible solution for signal conditioning.

5.1.7 Signal conditioning: possible solution

A new signal conditioning circuit has been designed in LTSpice using the following requirements:

- Both the positive and negative part of the mains waveform should be measurable;
- The voltage should be withing the linear region of the ADC (150 mV to 2450 mV);
- The amplitude of the waveform should be high enough to measure the signal with enough detail;
- The amplitude of the waveform should be low enough to show voltage surges;
- The resulting signal cannot get below ground to prevent damage to the ADC.

Concretely this would mean that a static voltage has to be added to the attenuated mains voltage (N$21 in Fig. 10). This would make the signal completely positive. Some way of controlling the amplification of this signal has to be implemented as well as something to keep the voltage above 0 V in all situations. A possible solution would involve three op-amps and a negative voltage source. The currently used LTC2055 can also be used for this application. It supports $\pm 5\ V$ supply voltage and has a slew rate of 0.5 $V/\mu s$ [21]. As the mains voltage is divided by 1000, this would equate to a maximum slew rate of 500 $V/\mu s$ for the mains voltage, which is sufficient.
The first op-amp stage (see Fig. 12) would act as an inverting voltage adder. This adds some multiple of the voltage of N$21 to some fraction (e.g. half) of the positive logic supply. This voltage then has to be inverted with a second op-amp stage before being fed to the ADC. All op-amps besides U4 (Fig. 12) have been connected to ground instead of V_{neg}, which should prevent V_{out} from being negative. A schematic of this can be seen in Fig. 12 with the important parameters in table 2.

Table 2: Important components of Fig. 12.

Identifier	Function or explanation
N$21	Mains voltage divided by 1000
V_{ac1}	Mains voltage
V_{out}	Conditioned mains voltage
V_{pos}	Positive logic supply (3.3v)
V_{neg}	Negative logic supply (-3.3v)
R_{12}, R_{13}	Resistor to set multiplication factor of N$21
$R_{13}, R_{14}, R_{15}, R_{16}$	Resistors to set offset voltage

The output voltage of the system can be calculated as follows:

$$V_{\text{out}} = \frac{R_{13}}{R_{14}} \frac{R_{16}}{R_{15} + R_{16}} V_{\text{pos}} + N\$21 \frac{R_{13}}{R_{12}} \quad (2)$$

For simplicity R_{17} and R_{18} are set to 10 $k\Omega$.
Depending on the component values, this system can be tuned to meet the requirements for different mains voltages. In Fig. 12 values are set to meet the requirements for a 230 V RMS system. Half of the logic supply voltage is added to twice the voltage of N$21. The output can be calculated as follows:

$$V_{\text{out}} = (1/2)V_{\text{pos}} + 2N\$21 \quad (3)$$

The plotted output signal can be seen in Fig. 13. As can be seen, the signal is within the linear region of the ADC, has reasonable amplitude and there is still some range left for transients.
In the end the actual mains voltage can be calculated from the sampled values using the following formula:

$$V_{\text{ac1}} = (V_{\text{out}} - \frac{R_{13}}{R_{14}} \frac{R_{16}}{R_{15} + R_{16}} V_{\text{pos}}) \frac{1000}{(R_{13}/R_{12})} \quad (4)$$

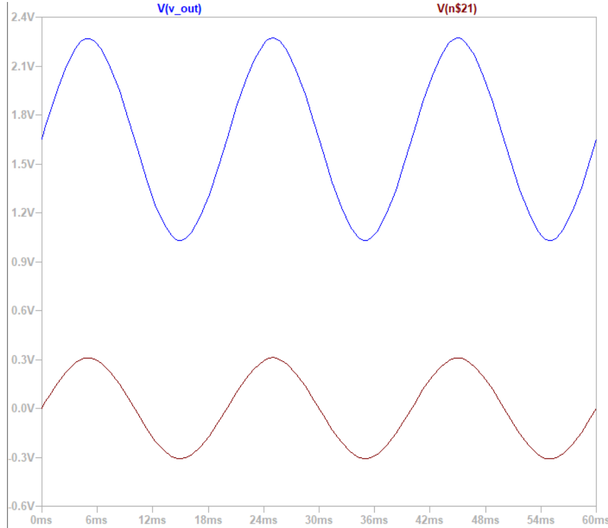

Figure 13: Input (black) and output (blue) voltage of the proposed system.

5.2 Data analysis using PC

Despite the setback regarding signal conditioning, some useful data analysis can still be prepared. Even though the mains voltage cannot be measured a generated signal can be used on an external ESP32-WROOM-32E. This way it can still be confirmed if the code for the ADC is correct and measurements have no unexpected artifacts.
During this analysis the signal generated by the HP 33120A are used, just like in Sec. 5.1.5. The studied signal is again a 10 kHz sine wave with $V_{PP} = 2\ V$ and an offset of 1400 mV. It will be applied to and measured by pin 8 of the ESP32 to mimic the conditions of the PQ monitor as good as possible [15]. Data will again be logged to the serial monitor and will be saved as a csv file. The format of this file is very simple: one raw 12 bit value (0-4095) per line. This value can be converted to a voltage using the following formula:

$$V_{\text{meas}} = \frac{3.3\ V_{\text{ADC}}}{4095} \qquad (5)$$

Here, V_{ADC} indicates the raw value from the ADC. To assess the usability of the signals, static voltages, the FFT magnitude spectrum and the 10 kHz time-signal will be studied. Three different sampling rates, 192 kHz, 384 kHz and 768 kHz, are compared. This is all done in MATLAB.

Time signal

To compare the different sampling rates, signals are plotted in Fig. 14. A measurement using a Rohde und Schwarz RTB2002 oscilloscope is also shown as a reference it is acquired by exporting the current display data of the scope to a csv. One column has the time information, the other column the voltage.

It can be seen that all signals have some offset when compared to the reference. In the higher and lower end there seems to be a bigger difference between measurement and actual value. Around the median (1400 mV) the measurements seem most accurate. This is applicable for all sampling rates. Besides these facts and that sampling rates are as expected not much can be concluded from the time signals alone.

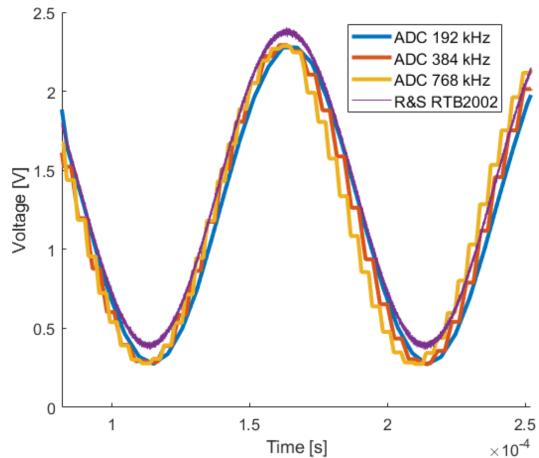

Figure 14: Measured signal plotted against signal from R&S RTB2002.

FFT magnitude spectra

To study noise performance the FFT magnitude spectra are compared. Fig. 15 shows all FFTs to their maximum bandwidth[3]. They have all been calculated using MATLAB from the time signals, except for the FFT by the oscilloscope (R&S RTB2002). This FFT has been made by exporting the current display information to a csv file again. In this case, one column has the frequency axis, the other column the magnitude.
From Fig. 15 it can be seen that all sampling rates show the 10 kHz peak very well. All FFTs look clean across most of their bandwidth, except for two peaks at around two third the bandwidth. They occur for every sampling rate.
The reason of these peaks is unclear to me. One way of explaining the peaks could be the frequency of the difference of static voltage measurements. Measuring a static voltage does not always return the same value due to the low precision of the ADC. This will be studied in more detail.
Lastly the magnitude plots will also be plotted using the MATLAB signal analyzer tool and zoomed in to better see the difference between the FFTs (Fig. 16). From the two Fig. 15 and 16 it can be concluded that higher sampling rates lead to lower noise levels.

[3]The FFT of the RTB2002 is limited to the maximum bandwidth of the ADC signals.

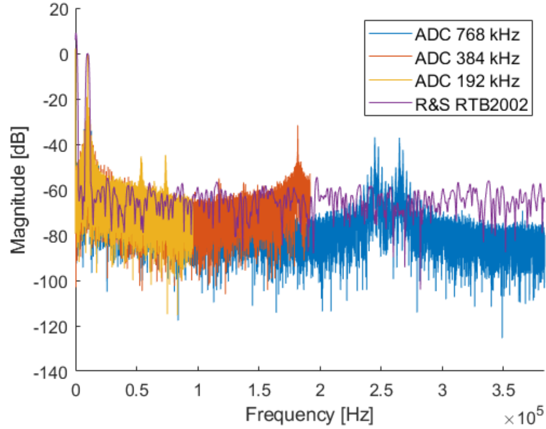

Figure 15: FFT magnitude spectrum of the ADC signals, using three sampling rates.

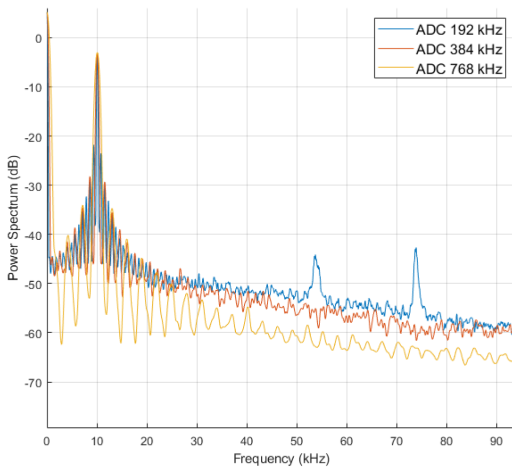

Figure 16: FFT magnitude spectrum: 0 to 96 kHz.

Static voltage measurements

To check performance (accuracy and precision) two voltages are considered. One voltage in the center of the linear region (at 1.25 V) and one measurement close to the non-linear region (2.5 V). Fig. 17 shows an example of the different values that are measured for an input of 1.250 V. The voltages are verified using an Agilent 34410A multimeter and will be called the 'reference'. It can be seen that the voltage readings of the ADC are not very precise ('stable') and that they are also not accurate. The limited precision is what I thought to be the reason of the two high frequency peaks in the FFTs. However, after performing an FFT on the static voltage no peaks were found. This means my theory is probably untrue. It is however good to keep in mind that the ADC has to be calibrated (due to offset) and multisampled if precise voltage analysis is your goal.
To elaborate on the precision and accuracy two plots are shown below (Fig. 18 and 19). They show the PDF of a voltage reading of 1.250 V and 2.507 V based on 32768 samples. Each voltage bin represents one of the 4096 possible measurement values. To compare the probability properties at the three different sampling rates, several metrics are shown in table 3 and 4. From these metrics it can be seen that the sampling rate does not deteriorate the performance regarding precision or accuracy. Standard deviation, which represents precision, is very similar for all sampling rates. The lower the standard deviation, the more precise the measurement. In fact, measurements at the lowest frequency show somewhat worse performance compared to higher frequencies. The same can be said about accuracy compared to frequency.
When comparing the two voltages it can be seen that the precision is just slightly higher for the 1.25 V measurement. The opposite is true regarding accuracy, where the expectation would be the right measure. For the 1.25 V measurement the difference with the reference value is 0.117 V, while it is 0.103 V in the 2.507 V case.
All in all, the ADC seems usable for high frequency operation and analysis. As long as the two peaks, shown in Fig. 15, are taken into account. For voltage measurements the accuracy and precision have to be taken into account. Some calibration method, e.g. with a reference voltage, would be preferable.

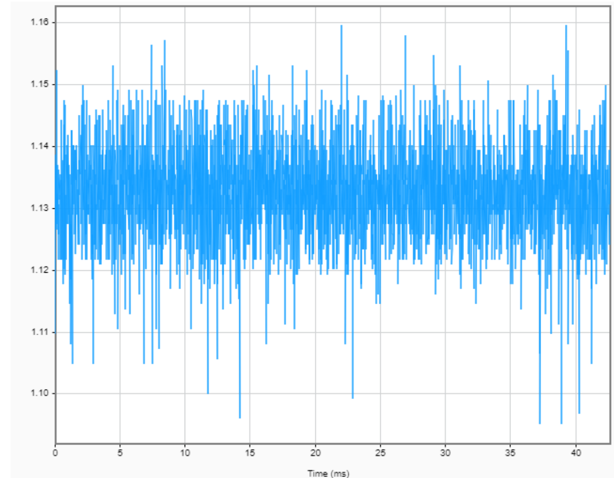

Figure 17: 1.25 volts measured by the ESP32 @ 768 kHz.

Table 3: Measurement of 2.5 volts at different sampling rates (N = 32768).

Sample speed	Expectation [V]	Stan. dev. σ
192kHz	2.4036	0.0053
384kHz	2.4048	0.0051
768kHz	2.4045	0.0050

Table 4: Measurement of 1.25 volts at different sampling rates (N = 32768).

Sample speed	Expectation [V]	Stan. dev. σ
192kHz	1.1329	0.0045
384kHz	1.1331	0.0043
768kHz	1.1330	0.0044

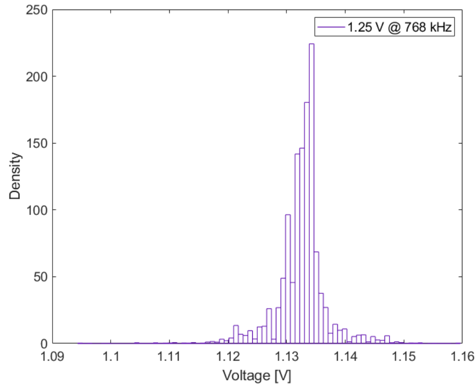

Figure 18: PDF of 1.25 volts measured by the ESP32 @ 768 kHz.

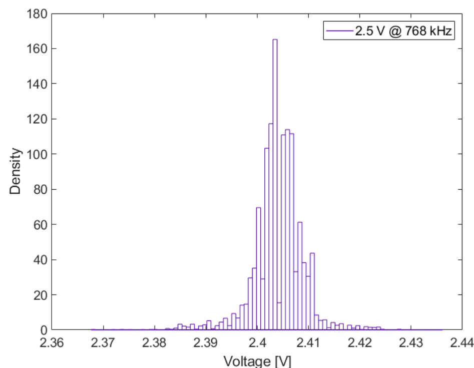

Figure 19: PDF of 2.5 volts measured by the ESP32 @ 768 kHz.

6 Future work

The final goal would be to do all DSP on the microcontroller and store processed data on the SD-card. Some work has been done regarding this. The work is non-conclusive but still included to help with future work.

6.1 Analysis on microcontroller

To perform the FFT on the microcontroller some research has been done. At first it was tried to implement ArduinoFFT [22]. This was not successful. An option to consider would be esp-dsp [23]. It is a library especially designed for use with the ESP32 and has FFT functions built in. Some notes on performance can be found in Appendix C.

6.2 Data storage

An attempt was made to implement data storage as described in the method. The SD card component of the original project by Roelof Grootjans and Niek Moonen was expanded [2].
Storage did not work in combination with the I2S sampling operation. Possible problems could be problems with the I2S clock configuration of my esp-idf version. APLL was also turned of to rule out the I2S clock source was the problem.
For future research I would advise to install the newest version of esp-idf (and update esp-idf if it is already installed on your PC). This could solve compatibility problems between components.
Possibly data could even be uploaded to a server to obviate the need to get the SD-card from the device.

7 Conclusion

Based on the findings of this paper the discussed PQ monitor seems usable for measuring high frequency data with some adjustments. No complete system was found to accomplish these measurements. However, all components leading to a working system have at least been partially researched.
It was first established that the minimum sampling rate had to be 500 kHz to be able to troubleshoot a power grid [1]. The cheap PQ monitor could reach this speed using I2S sampling. Research revealed that, as long as you stay within the linear region of the ADC, these samples are usable for further investigation. The signal conditioning of the line voltage has to be changed as discussed to get it into this linear range (150-2450mV).
A good representation of this voltage data would be the FFT. This shows the frequency disturbances over the whole measurable bandwidth. Having a lot of FFT data can show the change over time and allows to find out more about sources of PQ/EMI problems. RMS, minimum and maximum voltage would also be good to know in a very small timespan. This can tell way more about saggs, swells and impulses than measurements using the original PQ monitor. Sadly, these measurements were not performed on an actual grid.
All in all research seems promising. With more time and the implementation of different signal conditioning the PQ monitor can be expanded to do the measurements. At least raw measurement data can be saved to an SD card and analyzed using MATLAB. With the use of ESP-DSP the data could even be processed on the PQ monitor.

References

[1] Jako Kilter et al. *Guidelines for Power Quality Monitoring - Results from CIGRE/-CIRED JWG C4.112*. 2014.

[2] Roelof Grootjans and Niek Moonen. "Design of Cost-Effictive Power Quality and EMI Sensor for Multinode Network". In: *Letters on elektromagnetic compatibility practice and applications* 5.4 (Dec. 2023), pp. 131–136.

[3] Alexander Matthee, Niek Moonen, and Frank Leferink. "Synchronous Multipoint Low-Frequency EMI Measurement and Applications". In: *Letters on elektromagnetic compatibility practice and applications* 4.4 (Dec. 2022), pp. 120–124.

[4] Frank Leferink, Cees Keyer, and Anton Melentjev. "Static Energy Meter Errors Caused by Conducted Electromagnetic Interference". In: *IEEE Electromagnetic Compatibility Magazine* 5.4 (2016), pp. 49–55.

[5] Anna Dijkman. *Vastgelopen*. Apr. 2024. URL: https://fd.nl/opinie/1512545/vastgelopen.

[6] Puck Voorhoeve. *Overbelast stroomnet raakt bedrijven en woningbouw*. May 2022. URL: https://nos.nl/nieuwsuur/artikel/2427174-overbelast-stroomnet-raakt-bedrijven-en-woningbouw.

[7] Alexander Kusko and Marc T. Thompson. *Power Quality in Electrical Systems*. McGraw-Hill, 2007.

[8] Espressif Systems (Shanghai) Co. Ltd. *Analog to Digital Converter (ADC)*. URL: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/esp32/api-reference/peripherals/adc.html.

[9] Patrick. *ESP32 ADC Speed — LabFruits — labfruits.com*. https://www.labfruits.com/esp32-adc-speed/. [Accessed 17-06-2024]. 2018.

[10] Espressif Systems (Shanghai) Co. Ltd. *ADC DMA Example (commit from 2022)*. [Accessed 27-06-2024]. 2022. URL: https://github.com/espressif/esp-idf/tree/release/v4.4/examples/peripherals/adc/dma_read/main.

[11] elidupree et al. *ESP32 ADC DMA generates samples way too fast? (IDFGH-7285)*. [Accessed 28-06-2024]. 2022. URL: https://github.com/espressif/esp-idf/issues/8874.

[12] Espressif Systems. *ESP32 Series datasheet (Version 4.5)*. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. [Accessed 17-06-2024]. 2024.

[13] Cypress Semiconductor Corporation. *Inter-IC Sound Bus (I2S)*. [Accessed 19-06-2024]. 2012. URL: https://www.infineon.com/dgdl/Infineon-Component_I2S_V2.30-Software%20Module%20Datasheets-v02_07-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ea35b2125c8.

[14] Philips Semiconductors. *I2S bus specification*. [Accessed 19-06-2024]. 1996. URL: https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf.

[15] Unknown. *UNIT 00001.8 PQ-Monitor v52 (PQ monitor datasheet)*. 2023.

[16] Mehran Maleki. *ESP32 Pinout Reference: A Comprehensive Guide*. [Accessed 19-06-2024]. URL: https://electropeak.com/learn/full-guide-to-esp32-pinout-reference-what-gpio-pins-should-we-use/.

[17] Espressif Systems. *Inter-IC Sound (I2S)*. [Accessed 19-06-2024]. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/i2s.html.

[18] EEVBlog. *Topic: Agilent 33120A outputs double output voltage*. [Accessed 21-06-2024]. 2021. URL: https://www.eevblog.com/forum/repair/agilent-33120a-outputs-double-output-voltage/.

[19] Lulu's blog. *Linearity of the ESP32 ADC*. [Accessed 20-06-2024]. 2023. URL: https://lucidar.me/en/esp32/linearity-of-the-esp32-adc/.

[20] Analog Devices. *Energy Metering IC with Autocalibration ADE9514A*. [Accessed 20-06-2024]. 2018. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ade9153a.pdf.

[21] Linear Technology Corporation. *LTC2054/LTC2055 Single/Dual Micropower Zero-Drift Operational Amplifiers*. [Accessed 26-06-2024]. 2004. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/20545fc.pdf.

[22] kosme. *arduinoFFT - Fast Fourier Transform for Arduino*. [Accessed 23-06-2024]. 2017-2024. URL: https://github.com/kosme/arduinoFFT.

[23] Espressif. *DSP library for ESP-IDF*. [Accessed 23-06-2024]. 2024. URL: https://github.com/espressif/esp-dsp.

[24] Espressif Systems (Shanghai) Co. Ltd. *I2S Built-in ADC/DAC Example (commit from 2022)*. [Accessed 27-06-2024]. 2022. URL: https://github.com/espressif/esp-idf/tree/release/v4.4/examples/peripherals/i2s/i2s_adc_dac/main.

[25] Espressif Systems (Shanghai) Co. Ltd. *Espressif DSP Library Benchmarks*. [Accessed 23-06-2024]. 2019-2024. URL: https://docs.espressif.com/projects/esp-dsp/en/latest/esp32/esp-dsp-benchmarks.html.

8 Appendix

A Code adc_dma

This code is based on the adc_dma_example from esp-idf version 4.4 [10].

```
#include <string.h>
#include <stdio.h>
#include "sdkconfig.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "driver/adc.h"

#define ADC_RESULT_BYTE     2                                               // Size
    of ADC result in bytes
#define ADC_CONV_LIMIT_EN   1                          //For ESP32, this should always be
    set to 1
#define ADC_CONV_MODE       ADC_CONV_SINGLE_UNIT_1  //ESP32 only supports ADC1 DMA
    mode
#define ADC_OUTPUT_TYPE     ADC_DIGI_OUTPUT_FORMAT_TYPE1
#define TIMES 1024
#define GET_UNIT(x)        ((x>>3) & 0x1)
#define SAMPLING_RATE      83333                                        // Sampling
    rate. Max expected value is 83333

// Define channels
static uint16_t adc1_chan_mask = BIT(7);
static uint16_t adc2_chan_mask = 0;
static adc_channel_t channel[1] = {ADC1_CHANNEL_4};

static const char *TAG = "ADC DMA";

// Define functions
static void continuous_adc_init(uint16_t adc1_chan_mask, uint16_t adc2_chan_mask,
    adc_channel_t *channel, uint8_t channel_num);

void app_main(void){
    //Main app.
    esp_err_t ret;
    uint32_t ret_num = 0;
        // Log to serial monitor for debugging
    ESP_LOGI(TAG, "HERE");
    uint8_t result[TIMES] = {0};
        // Initialize memory
    memset(result, 0xcc, TIMES);

    //Initialize the ADC driver
    continuous_adc_init(adc1_chan_mask, adc2_chan_mask, channel, sizeof(channel) /
        sizeof(adc_channel_t));
    adc_digi_start();

    // Start loop
    bool go = true;
    while(go){
        //Loop
```

```
46          ESP_LOGI(TAG, "IN LOOP");
47              ret = adc_digi_read_bytes(result, TIMES, &ret_num, ADC_MAX_DELAY);
48              ESP_LOGI(TAG, "READ");
49              if (ret == ESP_ERR_INVALID_STATE){
50                  ESP_LOGI(TAG, "INVALID STATE");
51              }else if (ret == ESP_ERR_TIMEOUT){
52                  ESP_LOGI(TAG, "TIMEOUT");
53              }else{
54                  ESP_LOGI(TAG, "DONE1");
55                  vTaskDelay(1);
56              }
57          ESP_LOGI(TAG, "DONE");
58          ESP_LOGI("TASK:", "ret is %x, ret_num is %d", ret, ret_num);
59              for (int i = 0; i < ret_num; i += ADC_RESULT_BYTE) {
60                  adc_digi_output_data_t *p = (void*)&result[i];
61                  ESP_LOGI(TAG, "Unit: %d, Channel: %d, Value: %d", 1, p->type1.channel,
                ↪   p->type1.data);
62              }
63          go = false;
64      }
65      //Stop digi read
66      adc_digi_stop();
67      // Deinitialize
68      adc_digi_deinitialize();
69
70  }
71
72  static void continuous_adc_init(uint16_t adc1_chan_mask, uint16_t adc2_chan_mask,
    ↪   adc_channel_t *channel, uint8_t channel_num)
73  {
74      adc_digi_init_config_t adc_dma_config = {
75          .max_store_buf_size = 1024,
76          .conv_num_each_intr = TIMES,
77          .adc1_chan_mask = adc1_chan_mask,
78          .adc2_chan_mask = adc2_chan_mask,
79      };
80      ESP_ERROR_CHECK(adc_digi_initialize(&adc_dma_config));
81
82      adc_digi_configuration_t dig_cfg = {
83          .conv_limit_en = ADC_CONV_LIMIT_EN,
84          .conv_limit_num = 250,
85          .sample_freq_hz = SAMPLING_RATE,
86          .conv_mode = ADC_CONV_MODE,
87          .format = ADC_OUTPUT_TYPE,
88      };
89
90      adc_digi_pattern_config_t adc_pattern[SOC_ADC_PATT_LEN_MAX] = {0};
91      dig_cfg.pattern_num = channel_num;
92      for (int i = 0; i < channel_num; i++) {
93          uint8_t unit = GET_UNIT(channel[i]);
94          uint8_t ch = channel[i] & 0x7;
95          adc_pattern[i].atten = ADC_ATTEN_DB_2_5;
96          adc_pattern[i].channel = ch;
97          adc_pattern[i].unit = unit;
98          adc_pattern[i].bit_width = SOC_ADC_DIGI_MAX_BITWIDTH;
99
100         ESP_LOGI(TAG, "adc_pattern[%d].atten is :%x", i, adc_pattern[i].atten);
101         ESP_LOGI(TAG, "adc_pattern[%d].channel is :%x", i, adc_pattern[i].channel);
```

```
102         ESP_LOGI(TAG, "adc_pattern[%d].unit is :%x", i, adc_pattern[i].unit);
103     }
104     dig_cfg.adc_pattern = adc_pattern;
105     ESP_ERROR_CHECK(adc_digi_controller_configure(&dig_cfg));
106 }
```

B Code i2s_adc

This code is based on the i2s_adc_dac example from esp-idf version 4.4 [24]. The code can also be used to calculate the 1024 point FFT of a test signal.

```
1  #include <stdio.h>
2  #include <stdlib.h> // for DSP
3  #include <string.h>
4  #include "freertos/FreeRTOS.h"
5  #include "freertos/task.h"
6  #include "esp_system.h" // for DSP
7  #include "esp_err.h"
8  #include "esp_log.h"
9  #include "driver/i2s.h"
10 #include "driver/adc.h"
11 #include "esp_rom_sys.h"
12 #include <esp_task_wdt.h> // to change wdt seconds to allow for data transfer to pc
13 #include "driver/spi_master.h"
14 // All includes under this text = DSP
15 #include "soc/gpio_struct.h"
16 #include "driver/gpio.h"
17 #include "driver/uart.h"
18 #include "soc/uart_struct.h"
19 #include <math.h>
20 // See
   ↪  https://docs.espressif.com/projects/esp-dsp/en/latest/esp32/esp-dsp-benchmarks.html
   ↪  for benchmarks (w.r.t. compiler settings)
21 #include "esp_dsp.h"
22
23 // Choose to sample or to do a 1024 point FFT on sample data
24 #define SAMPLING 1
25 //#define DSP 1
26
27 static const char* TAG = "ad/da";
28 #define V_REF   1100
29 #define ADC1_TEST_CHANNEL (ADC1_CHANNEL_4)
30
31 //i2s number
32 #define I2S_NUM (0)
33 //i2s sample rate
34 #define I2S_SAMPLE_RATE (384000) // multiples of 48k seem to work best, 384k (=768
   ↪  kHz) seems to be the max w/o bugs
35 //i2s data bits
36 #define I2S_SAMPLE_BITS (16)
37 #define BUF_LEN 1024
38 #define BUF_CNT 2
39 //i2s read buffer length
40 #define I2S_READ_LEN (I2S_SAMPLE_BITS * BUF_LEN)
41 //i2s data format
42 #define I2S_FORMAT  (I2S_CHANNEL_FMT_RIGHT_LEFT)
43 //i2s channel number
```

```
44  #define I2S_CHANNEL_NUM ((I2S_FORMAT < I2S_CHANNEL_FMT_ONLY_RIGHT) ? (2) : (1))
45  //i2s built-in ADC unit
46  #define I2S_ADC_UNIT ADC_UNIT_1
47  //i2s built-in ADC channel
48  #define I2S_ADC_CHANNEL ADC1_CHANNEL_4
49
50  //define samples to record
51  #define RECORD_SIZE 8 * I2S_READ_LEN
52
53  //Defines from DSP example
54  #define N_SAMPLES 1024
55  int N = N_SAMPLES;
56  // Input test array
57  __attribute__((aligned(16)))
58  float x1[N_SAMPLES];
59  __attribute__((aligned(16)))
60  float x2[N_SAMPLES];
61  // Window coefficients
62  __attribute__((aligned(16)))
63  float wind[N_SAMPLES];
64  // working complex array
65  __attribute__((aligned(16)))
66  float y_cf[N_SAMPLES * 2];
67  // Pointers to result arrays
68  float *y1_cf = &y_cf[0];
69  float *y2_cf = &y_cf[N_SAMPLES];
70
71  // Sum of y1 and y2
72  __attribute__((aligned(16)))
73  float sum_y[N_SAMPLES / 2];
74
75
76  void i2s_init(void){
77      int i2s_num = I2S_NUM;
78      i2s_config_t i2s_config = {
79          .mode = I2S_MODE_MASTER | I2S_MODE_RX | I2S_MODE_TX | I2S_MODE_DAC_BUILT_IN |
            ↪  I2S_MODE_ADC_BUILT_IN,
80          .sample_rate =  I2S_SAMPLE_RATE,
81          .bits_per_sample = I2S_SAMPLE_BITS,
82          .communication_format = I2S_COMM_FORMAT_STAND_MSB,
83          .channel_format = I2S_FORMAT,
84          .intr_alloc_flags = 0,
85          .dma_buf_count = BUF_CNT,
86          .dma_buf_len = BUF_LEN,
87          .use_apll = 1,
88       };
89      //start driver
90      i2s_driver_install(i2s_num, &i2s_config, 0, NULL);
91      // init ADC pad
92      i2s_set_adc_mode(I2S_ADC_UNIT, I2S_ADC_CHANNEL);
93  }
94
95  // Important function: translates the I2S signal to an integer from 0 to 4095
96  void example_disp_buf(uint8_t* buf, int length)
97  {
98      printf("======\n");
99      for (int i = 0; i < length; i = i+2) {
100                 // Do bit shift and or operation (explained in paper)
```

```
101             uint16_t combined = ((uint16_t)(buf[i+1]) << 8) | buf[i];
102             // Mask to ensure it is 12-bit
103             combined &= 0x0FFF;
104             printf("%d\n", combined);
105         }
106         printf("======\n");
107     }
108 
109     void i2s_adc(void*arg){
110             // Print to console for debugging
111         printf("running task\n");
112         int i2s_read_len = I2S_READ_LEN;
113         int currentPos = 0;
114         size_t bytes_read;
115         char* i2s_read_buff = (char*) calloc(i2s_read_len, sizeof(char));
116         char* i2s_buff = (char*) calloc(RECORD_SIZE, sizeof(char));
117         i2s_adc_enable(I2S_NUM);
118         while(currentPos < RECORD_SIZE){
119             //read data from i2s ADC
120             i2s_read(I2S_NUM, (void*) i2s_read_buff, i2s_read_len, &bytes_read,
                ↪   portMAX_DELAY);
121             memcpy(i2s_buff + currentPos, i2s_read_buff, i2s_read_len);
122             currentPos += i2s_read_len;
123         }
124         i2s_adc_disable(I2S_NUM);
125             // Error in code: displayed both buffers, which is what led to the NaN errors
                ↪   in MATLAB. Resolved by commenting out the bad code.
126         //example_disp_buf((uint8_t*) i2s_read_buff, i2s_read_len);
127             example_disp_buf((uint8_t*) i2s_buff, RECORD_SIZE);
128             // Free memory
129         free(i2s_read_buff);
130         i2s_read_buff = NULL;
131         free(i2s_buff);
132         i2s_buff = NULL;
133         vTaskDelete(NULL);
134     }
135 
136     void app_main(void){
137         //main app
138         // change wdt to 120 seconds for serial communication
139         esp_task_wdt_init(120, false);
140         esp_log_level_set("I2S", ESP_LOG_INFO);
141 
142             // Tasks for sampling
143         #ifdef SAMPLING
144         i2s_init();
145         xTaskCreate(i2s_adc, "i2s_adc", 1024 * 2, NULL, 5, NULL);
146         #endif
147 
148             //Tasks for DSP
149         #ifdef DSP
150         esp_err_t ret;
151         ESP_LOGI(TAG, "Start Example.");
152         ret = dsps_fft2r_init_fc32(NULL, CONFIG_DSP_MAX_FFT_SIZE);
153         if (ret  != ESP_OK) {
154             ESP_LOGE(TAG, "Not possible to initialize FFT. Error = %i", ret);
155             return;
156         }
```

```
157
158     // Generate hann window
159     dsps_wind_hann_f32(wind, N);
160     // Generate input signal for x1 A=1 , F=0.1
161     dsps_tone_gen_f32(x1, N, 1.0, 0.16,  0);
162     // Generate input signal for x2 A=0.1,F=0.2
163     dsps_tone_gen_f32(x2, N, 0.1, 0.2, 0);
164
165     // Convert two input vectors to one complex vector
166     for (int i = 0 ; i < N ; i++) {
167         y_cf[i * 2 + 0] = x1[i] * wind[i];
168         y_cf[i * 2 + 1] = x2[i] * wind[i];
169     }
170     // FFT
171     unsigned int start_b = dsp_get_cpu_cycle_count();
172     dsps_fft2r_fc32(y_cf, N);
173     unsigned int end_b = dsp_get_cpu_cycle_count();
174     // Bit reverse
175     dsps_bit_rev_fc32(y_cf, N);
176     // Convert one complex vector to two complex vectors
177     dsps_cplx2reC_fc32(y_cf, N);
178
179     for (int i = 0 ; i < N / 2 ; i++) {
180         y1_cf[i] = 10 * log10f((y1_cf[i * 2 + 0] * y1_cf[i * 2 + 0] + y1_cf[i * 2 + 1]
            ↪  * y1_cf[i * 2 + 1]) / N);
181         y2_cf[i] = 10 * log10f((y2_cf[i * 2 + 0] * y2_cf[i * 2 + 0] + y2_cf[i * 2 + 1]
            ↪  * y2_cf[i * 2 + 1]) / N);
182         // Simple way to show two power spectrums as one plot
183         sum_y[i] = fmax(y1_cf[i], y2_cf[i]);
184     }
185
186     // Show power spectrum in 64x10 window from -100 to 0 dB from 0..N/4 samples
187     ESP_LOGW(TAG, "Signal x1");
188     dsps_view(y1_cf, N / 2, 64, 10,  -60, 40, '|');
189     ESP_LOGW(TAG, "Signal x2");
190     dsps_view(y2_cf, N / 2, 64, 10,  -60, 40, '|');
191     ESP_LOGW(TAG, "Signals x1 and x2 on one plot");
192     dsps_view(sum_y, N / 2, 64, 10,  -60, 40, '|');
193     ESP_LOGI(TAG, "FFT for %i complex points take %i cycles", N, end_b - start_b);
194
195     ESP_LOGI(TAG, "End Example.");
196     #endif
197
198 }
```

C A note on performance: esp-dsp

The performance of the FFT as well as the maximum length are dependent on the compiler/SDK settings. Using esp-idf menuconfig some settings can be changed to enhance the performance of the FFT.
In the component config, then DSP Library, the maximum FFT length can be set. Standard this is 4096 with a maximum of 32768. Furthermore the compiler can be optimized for speed or (program) size. This option can be found under partition table, then compiler options. Lastly it is important to check the actual CPU frequency under ESP32-specific, then CPU frequency (max = 240MHz).
To see the performance impact of the compiler settings a 1024 complex point FFT was performed. It should be noted that setting the configuration to "Optimize for performance" instead of "Optimize for size" made the FFT slower. When optimized for size it takes 139558 cycles to complete and 140525 otherwise. These findings are in line with the official benchmarks [25].