Otto-von-Guericke-University Magdeburg
Faculty of Electrical Engineering and Information Technology

Integrated Project Report

Ultrasonic time reversal acoustic sonar: Interfacing of an SD card, push-button and an ADC with the ESP32 MCU

Version 1.0, October 2021

submitted: November 2, 2021

by: Mohammad Taif Arif Shamsi
born on July 23, 1994
in Karachi, Pakistan

Abstract

A series of piezoelectric sensors were mounted outside a miniature cylindrical tank. An algorithm was to be developed which would utilize the data from these sensors in real-time and pinpoint the location of the cylindrical object inside the water tank. This project interfaced an LTC2314-14 analogue to digital converter with an ESP32 microcontroller. The ADC received the data from the piezoelectric sensor and the microcontroller made it into a useful format. The data was then saved into an SD card connected to the ESP32 microcontroller. A push button was also interfaced which provided the user with the functionality to start and stop the SPI transactions between the ESP32 MCU and the ADC. The success of this project provides a further opportunity to its scaling by incorporation of more piezoelectric sensors and finally the development of an algorithm for the localization of the object inside the water tank.

Declaration by the candidate

I hereby declare that this project work is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been marked.

The work has not been presented in the same or a similar form to any other testing authority and has not been made public.

Magdeburg, November 2, 2021

Contents

List of Acronyms

ADC Analogue to Digital Converter

ESP-IDF Espressif Internet of Things Development Framework

GPIO General-purpose input and output

IoT Internet of things

KB Kilo Bytes

KHz Kilo Hertz

mA mili Ampere

MCU Micro-controller Unit

MHz Mega Hertz

Msps Mega samples per second

PS Power Supply

SPI Serial Peripheral Interface

SRAM Static random-access memory

V Volt(s)

List of Figures

List of Tables

1 Introduction

This chapter describes the project in detail and also introduces the relevant hardware and software components used.

1.1 Project Overview

In industrial chemical tanks, vessels and columns, properties such as temperature, liquid level and pressure, may need to be monitored using various kinds of sensors.[19]

These sensors are typically wired or powered through batteries (in cases where wireless technologies are used). It is often advantageous to use wireless sensors over wired ones. Some advantages include: flexibility in deciding where to install the sensor, low cost, and the ease of commission. [2]

In a novel approach, wireless sensors are to be recharged through a wave reversal technique but would require the precise location of the sensor in real-time for it to be possible. For the location to be determined, a miniature cylindrical water tank is mounted with multiple piezoelectric sensors as in the figure 1.1 . The data from these sensors would be provided to the ADCs which would convert it into a useful digital format. An algorithm will take input data from all the sensors and provide the location of an object placed inside the tank. One example of such an object is shown in figure 1.2.

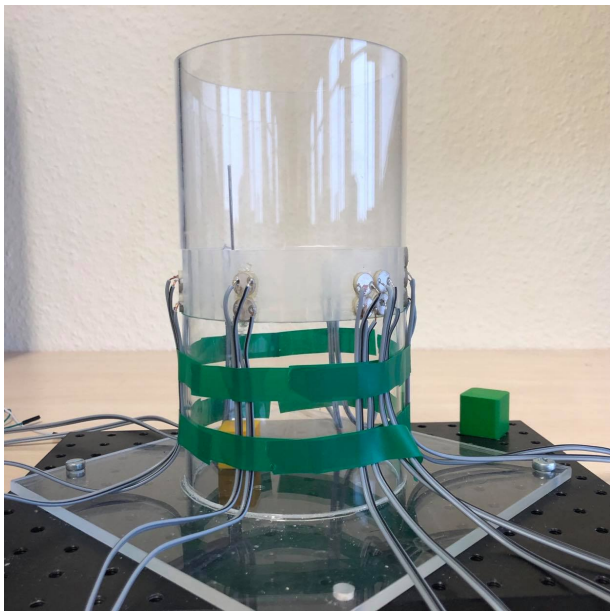


Figure 1.1: Miniature water tank with multiple piezoelectric sensors mounted outside the body for object detection. Apparatus provided by [5].

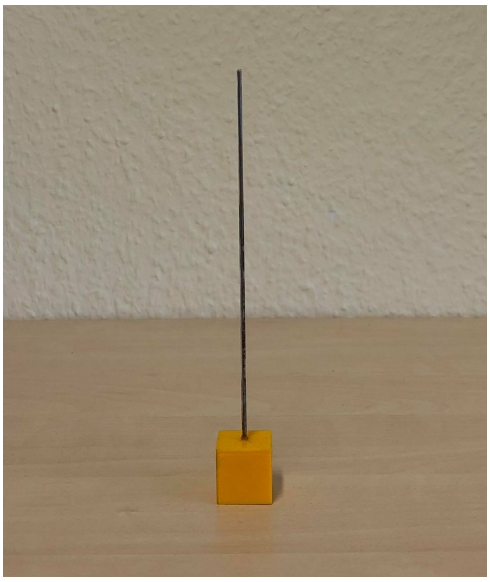


Figure 1.2: A cube object, whose location is to be determined in real-time. Apparatus provided by [5].

In this project, as a proof of concept, one ADC was interfaced with an ESP32 MCU. The ADC was given a number of different waveforms and the processed output was compared with the original input to see if the original wave could be reconstructed. The processed output was also saved into an SD card which was interfaced with the microcontroller. A push button was also interfaced with the MCU which provides the functionality to the user to start and stop the SPI transactions between the ESP32 MCU and the ADC. A block diagram of all the major blocks can be seen in figure 1.3.

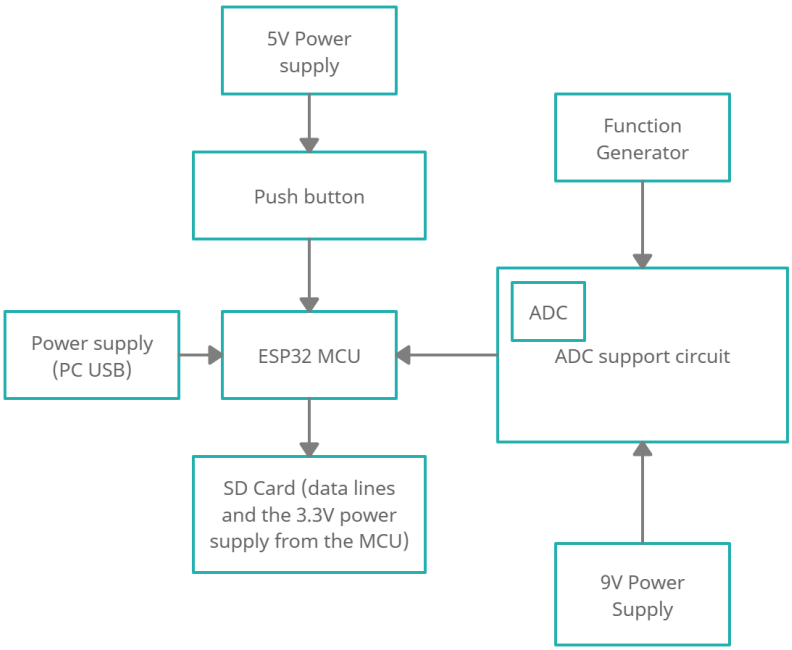


Figure 1.3: Project block diagram.

A schematic diagram for the project is also given in 1.4.

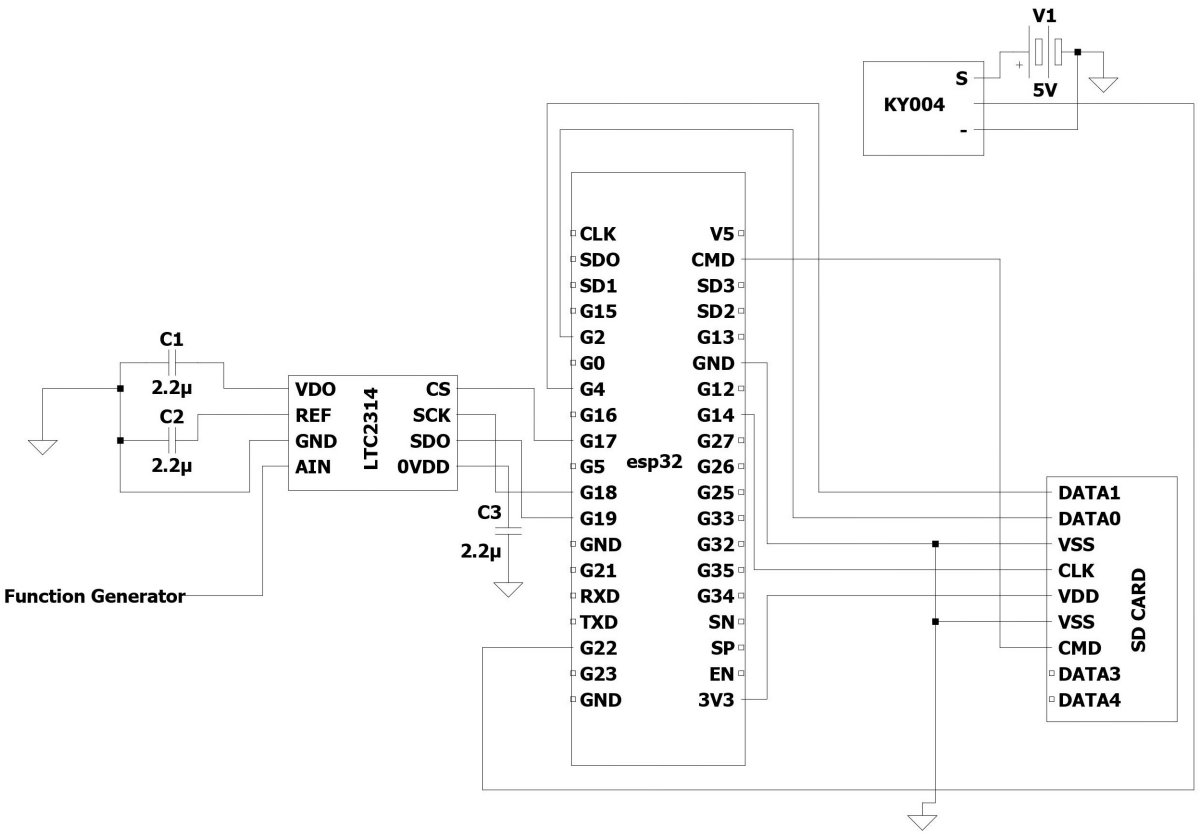


Figure 1.4: Schematic diagram of the project.

1.2 Hardware

1.2.1 ESP-32 Microcontroller

ESP32 DevKitC MCU by ESPRESSIF SYSTEMS was used in this project. It allows quick code testing and is easy to use on a breadboard. [17]

The development kit features a ESP-32-WROOM MCU module. It has two individual CPU cores the clock frequency of which can be set from 80 MHz to 240 MHz. Peripherals such as the SD card interface and the SPI interface is also available. A number of GPIO pins are also provided along with the ability to power using a 5V and a 3.3V power supply. It has 448KB of memory for booting and core functions, while 520KB of memory (SRAM) for data and instructions. The crystal oscillator used provides a frequency of 40 MHz. [18]

1.2.2 LTC2314-14 Analogue to Digital Converter

The LTC2314-14 provides a 14 bit resolution, and a throughput of 4.5Msps. The ADC is fairly power-efficient as only 6.2mA of current is drawn from the power supply. The chip also allows an SPI interface which we have used in this project. The ADC chip requires support circuitry which can be seen in figure 1.7. [3]

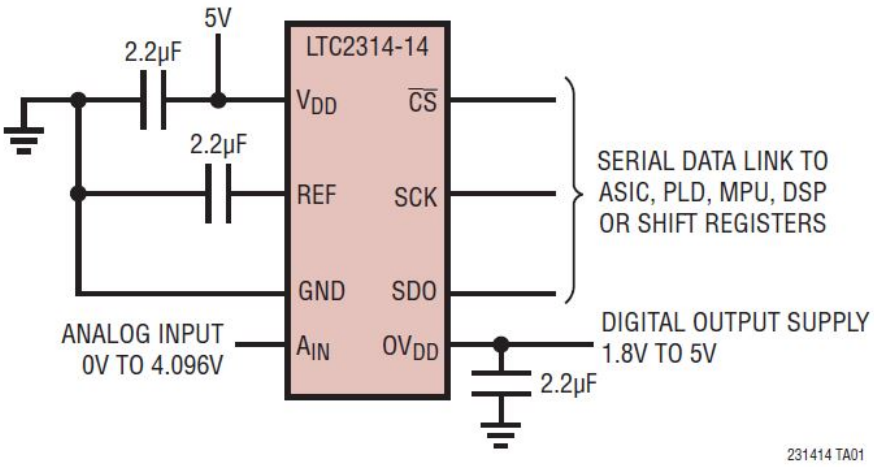
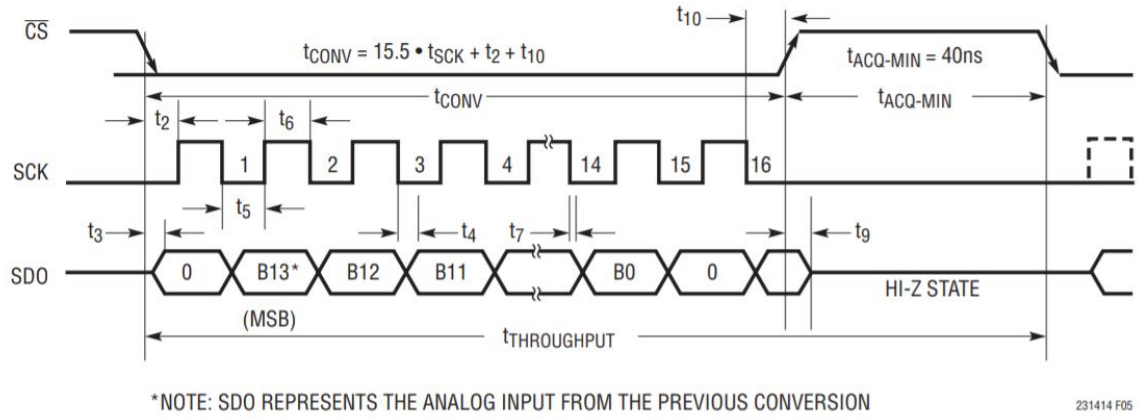


Figure 1.5: ADC support circuitry [3]

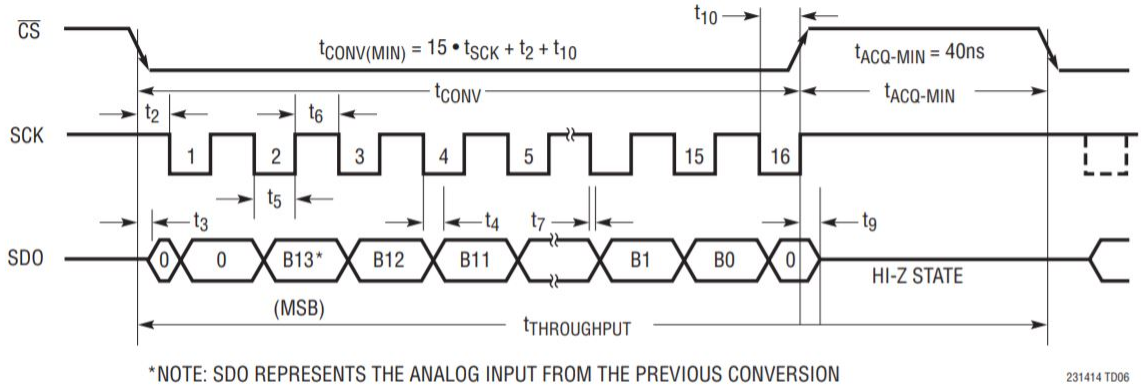
.

The ADC reacts differently to different incoming clocks as can be seen in figures 1.6a and 1.6b. In figure 1.6a, the clock polarity is 0 and the data is sampled on the first edge (rising edge) which gives the impression that the used SPI mode is 0.[16] [4]

In figure 1.6b the clock polarity is 1 and the data is sampled at the second edge (rising edge) which means the used SPI mode is 2 [16] [4]. There is also a continuous SPI mode available which is not discussed here. In this project, SPI mode 0 was used in consideration of the timings requirements displayed by figure 1.6a. The received data was further processed by deletion of the one leading and one trailing zero from the data.



(a) Serial interface timing diagram 1 [3].



(b) Serial interface timing diagram 2 [3].

Figure 1.6: Serial Interface timing diagrams for the ADC [3].

1.2.3 Joyit KY-004 Push button

The push button is used to control the flow of the program. Once pressing it will run the program once. It is powered by a 5V external power supply. The schematic diagram is shown figure 1.4.

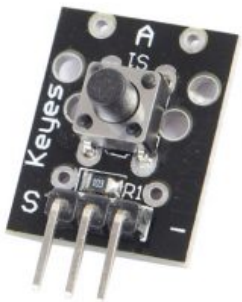


Figure 1.7: Joyit KY-004 Push button [14].

1.2.4 Connections

The Pin Configuration is as follows:

Pin Configuration					
ESP32 DevC	**LTC2314-14**	**SD Card**	**KY-004**	**5V PS**	**9V PS**
GPIO17	CS				
GPIO19	SCK				
GPIO18	SDO				
GPIO22			Middle Pin (unmarked)		
			S	+	
	+				+
GND	-		-	-	-
GPIO4		D1			
GPIO2		D0			
GND		GND			
GPIO14		CLK			
3.3V		VDD			
GND		GND			
GPIO15		CMD			
GPIO13		D3			
GPIO12		D2			

Table 1.1: Pin connections

1.3 Software

ESPRESSIF Systems provides the "Espressiv IoT Development Framework" also called the "esp-idf" for the development of application software for ESP32 MCUs. This is the official frameweork provided by the manufacturing company of the MCU. [11]

The integrated development environment being used is the VScode, with the ESP-IDF extension provided by the company.

Arduino also provides several libraries to program the ESP32 series MCUs. [9]

2 Platforms

2.1 Arduino

The following was provided by [5]: some changes have been made and the code has been commented for further understanding.

```
#include "esp32-hal-cpu.h"
#include "esp32-hal-spi.h"
#include "FS.h"
#include "SD.h"

#define HSPI_MISO   12
#define HSPI_MOSI   -1
#define HSPI_SCLK   14
#define HSPI_SS     15


//SPI clock is set to 80MHz which is the fastest possible
//on the ESP32 hardware.
static const int spiClk = 80000000; // 1 MHz

//Pointer to an SPIclass is null-initialized.
SPIClass * hspi = NULL;

//The number of continuous readings/ SPI transactions
//that need to be taken/ done with minimum delay.
const int Anz = 360;

//The value of the ADC is stored into this array.
int adcValue [Anz];

word MSBByte1[Anz] = {0};//Most significant BIT

//Pin number to which the push button is connected to.
```

```
int pushbutton = 26;


//Function for opening the file on the SD card
//with "appending" characteristic. This ensures that previous data
// is not overwritten.
void appendFile(fs::FS &fs, const char * path, const char * message) {

  Serial.printf("Appending to file: %s\n", path);

  File file = fs.open(path, FILE_APPEND);
  if (!file) {
    Serial.println("Failed to open file for appending");
    return;
  }
  if (file.println(message)) {
    Serial.println("Message appended");
  } else {
    Serial.println("Append failed");
  }
}

//The following codes runs one-time.
void setup()
{
  //The ESP32 CPU frequency is set to 240 MHz.
  setCpuFrequencyMhz(240);

  //The serial monitor baudrate is set to 115200
  Serial.begin(115200);

  //SPI object is created with HSPI pins.
  hspi = new SPIClass(HSPI);

 // SPI object is called into action setting the required pins.
  hspi->begin(HSPI_SCLK, HSPI_MISO, HSPI_MOSI, HSPI_SS);

  //slave select pin for the ADC is set as output.
  pinMode(HSPI_SS, OUTPUT); //HSPI SS
```

```
  //slave select pin set to HIGH/ no-read/no-write mode.
  digitalWrite(HSPI_SS, HIGH);

  // pin for the push button is internally pulled up because
  //the button is active-low.
  pinMode(pushbutton, INPUT_PULLUP);

  //The "If-statement" outputs an error if the SD card is undetected.
  if (!SD.begin())
  {
    Serial.println("Card Mount Failed");
    return;
  }
}

//The following code runs continuously.
void loop()
{
  //variable to store the push-button state.
  int buttonstate = digitalRead(pushbutton);

  //If the button is pressed, the reading starts and is
  //saved on an SD card.
  if (buttonstate == 0)

  {
    //SPI transactions begin.
    hspi->begin();

    //SPI clock, format and the mode is set for the transaction.
    hspi->beginTransaction(SPISettings(spiClk, MSBFIRST, SPI_MODE0));

    //The "for-loop" runs continuous transactions
    //with minimum delay for the number specified by Anz (Here: 360).
    for (int i = 0; i < Anz; i++)
    {
      //Slave-select pin is pulled low, so that the
      //reading can begin.
```

```
  digitalWrite(HSPI_SS, LOW);

  //16 bits are transferred from the ADC to the
  //ESP32 MCU and save in the array.
  MSBByte1[i] = hspi->transfer16(0);

  //Slave-select pin is pulled high so the transaction can end.
  digitalWrite(HSPI_SS, HIGH);
}
//This marks the end of the transactions.
hspi->endTransaction();
hspi->end();

//The saved values in the array are processed. This means
//removing the leading and the trailing zeros.
for (int i = 0; i < Anz; i++)
{
  // This moves the trailing zero
  MSBByte1[i] = MSBByte1[i] >> 1;

  //This removes the leading zero
  adcValue[i] = MSBByte1[i] & 0x3FFF;
}

int max = 0, reading, maxIndex = 0;
for (int i = 0; i < Anz; i++) {
  reading = adcValue[i]; // change number for pin you are using
  Serial.println(reading);
  if (reading > max) max = reading; maxIndex = i;
}

Serial.print("max :");
Serial.println(max);
Serial.print("maxIndex :");
Serial.println(maxIndex);

//String for storing incloming ADC values to a file.
String dataString = "";
```

```
    //The following for-loop saves the values read from the A
    //DC into a string array.
    for (int i = 0; i < Anz; i++)
    {
      //All the ADC values are saved into the "dataString" array.
      dataString += String (adcValue[i]) + " ";
    }
    //With one line of code, the data from the dataString array is
    //appended into the hello.txt file on the SD card.
    appendFile(SD, "/hello.txt", dataString.c_str());

    Serial.println(dataString);

    //A delay of 5 seconds is added after the saving.
    delay(5000);
  }

}
```

Compiling the code results in consecutive transaction time of around 1.1 micro seconds as can be seen in the figure 2.1.

The transaction delay is between the clear signal and the SPI-clock of the ESP32. It was suggested then to use the ESP32 IDF framework in an attempt to reduce this time delay so that the full throughput of 4.5Msps of the ADC can be utilized.

Figure 2.1: Consecutive SPI transactions delay.

The transaction delay is between the clear signal and the SPI-clock of the ESP32 as can be seen in the figures 2.2 and 2.3. It was suggested then to use the ESP32 IDF framework in an attempt to reduce this time delay so that the full throughput of 4.5Msps of the ADC can be utilized.

Figure 2.2: Delay between the end of the clock and the clear signal being pulled high.

Figure 2.3: Delay between the start of the clock after the clear signal is being pulled low.

2.2 ESP IDF

2.2.1 Environment Setup

First the VScode IDE was downloaded. Then the ESP IDF extention was installed through the extention library function. It was made sure that the C++ build tools were available before the extension installation. [6]

The speed of the CPU clock was also configured. The ESP-IDF SDK configuration editor option was selected which is located at the bottom of the screen. Then the keyword clock was searched in the appearing search bar.

2.2.2 External Library

In order to use external libraries the following steps were taken. Firstly, the library which is to be downloaded was placed inside the 'components' folder of the esp-idf library pathway. The name of the folder was renamed to whatever is described in the Github page from where the library is downloaded from. Failing to do so would have caused a compilation error as the required libraries would not have been found. Next, the example code to be used was imported it into the VScode and by opening it as a folder. In this case, the example code folder was renamed to "SPI". Another folder was created and renamed to "main", inside this example "SPI" folder which contains a main.cpp file (main source code file). In the main folder a CmakeLists.txt file was created with the following contents:

```
idf_component_register(SRCS "main.cpp"
```

```
                    INCLUDE_DIRS ".")
```

where main.cpp is the name of your source code file. Next, it was ensured that the "SPI" folder contained another CmakeLists.txt file with the following contents:

```
# The following five lines of boilerplate have to be in your
# project's CMakeLists in this exact order for cmake to work
# correctly
cmake_minimum_required(VERSION 3.5)


include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(SPI)
```

Where SPI is the name of the example folder.

The following is the content of the SPI file. The library is provided by [15]:

```
#include<stdio.h>
#include<stdint.h>
#include<freertos/FreeRTOS.h>
#include"freertos/task.h"
#include"esp_log.h"
#include"esp_err.h"
#include"SPIbus.hpp"

#define SPI_MODE 0
#define MISO_PIN 12
#define MOSI_PIN 13
#define SCLK_PIN 14
#define CS_PIN   15
#define SPI_CLOCK 10000000 // 1 MHz

extern "C" void app_main() {
    printf("SPIbus Example \n");
    fflush(stdout);

    SPI_t &mySPI = vspi;  // vspi and hspi are the default objects

    // Device handle for each slave should be created.
    spi_device_handle_t device;
    ESP_ERROR_CHECK( mySPI.begin(MOSI_PIN, MISO_PIN, SCLK_PIN));
```

```
    ESP_ERROR_CHECK( mySPI.addDevice(SPI_MODE, SPI_CLOCK, CS_PIN,
    &device));

    uint8_t buffer[6];
    while (1) {
        // This function reads 2 Bytes from the address 0x0B.
        ESP_ERROR_CHECK(mySPI readBytes(device, 0x0B, 2, buffer));
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }

    mySPI.removeDevice(device);
    mySPI.close();
    vTaskDelay(portMAX_DELAY);
}
```

As can be seen in the 2.4, there is no delay between the clear signal and the start of the clock cycle. This gives the motivation that theoretically this phenomena can also be promoted to multiple transactions with no delay which would utilize the whole throughput of the ADC.

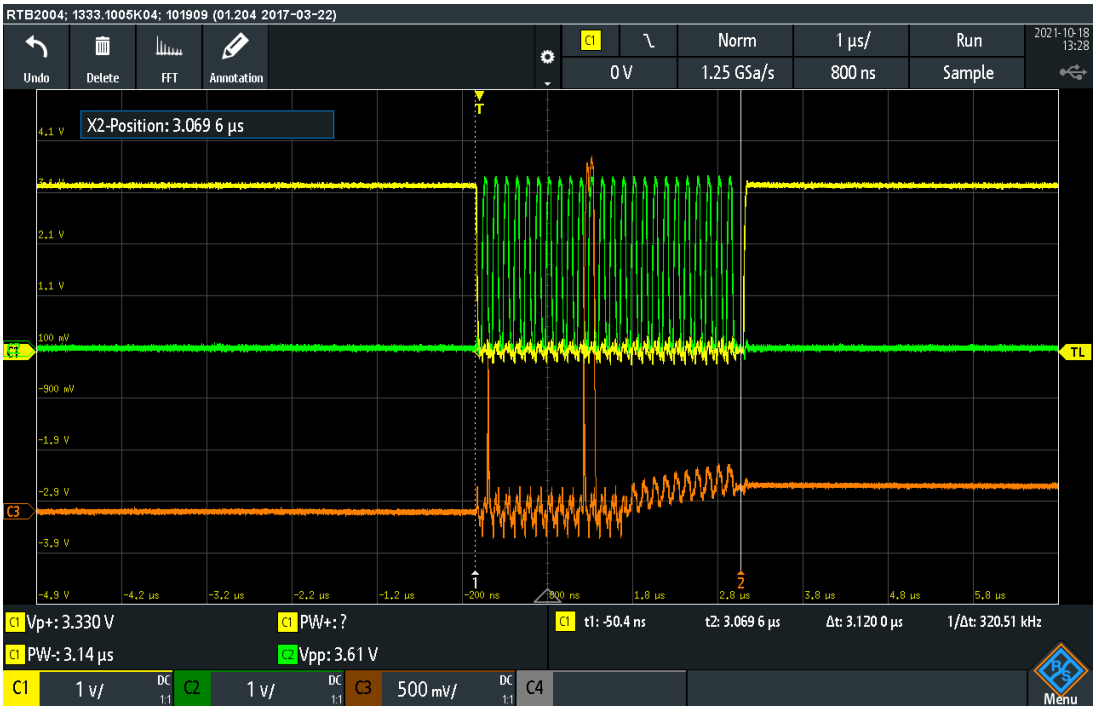


Figure 2.4: Single transaction of the example code.

However, by simply calling the transactions recursively through a for-loop, there appears to be a 10ms delay between each transactions which can be seen in the figure 2.5. This calls for a better method to improve the timing. At this moment, the ESP-IDF drivers come into play.

Figure 2.5: Consecutive transactions of the example code.

Limitations

The library although simple to use, does not give direct control of the data being received. The functions being called displays the data on the serial monitor which hinders the user from being able to manipulate the variables. Such as processing it to exclude the leading and the trailing zeros and saving it into the SD card. It also gives no control over if the user prefers polling method which significantly reduces the transaction time. These findings led to the use of drivers provided by the ESPRESSIV Systems.

3 Software Components

3.1 Working Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "driver/spi_master.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "esp_err.h"

#include <sys/unistd.h>
#include <sys/stat.h>
#include "esp_vfs_fat.h"
#include "driver/sdspi_host.h"
#include "sdmmc_cmd.h"
#include "sdkconfig.h"
#include "driver/sdmmc_host.h"

static const char *TAG1 = "ADC";

//There are two HOSTs available, the VSPI host is used
// due to the flexibility of the pins.
#define HOST VSPI_HOST
// The MISO pin is defined, which gets the data from the slave.
#define PIN_NUM_MISO 19
// MOSI pin is not used as no command bits are
//required by the ADC.
#define PIN_NUM_MOSI -1
#define PIN_NUM_CLK 18
// The CS Pin is manually defined so it is declared
```

```
// here as not in use.
#define PIN_NUM_CS -1

//The clock is defined a max of 80 mega Hertz.
#define CLOCK 80 * 1000 * 1000
#define MOUNT_POINT "/sdcard"
// Pin for the CS signal defined.
#define GPIO_OUTPUT_IO_0 17
// Pin for the Pushbutton defined.
#define GPIO_INPUT_IO_0 22
// CS signal pin masked (register position).
#define GPIO_OUTPUT_PIN_SEL (1ULL << GPIO_OUTPUT_IO_0)
// Push button pin masked (register position).
#define GPIO_INPUT_PIN_SEL (1ULL << GPIO_INPUT_IO_0)

void app_main(void)

{
    //variable to store and check the state of the pin
    uint8_t pin = 1;
    //variable to check if button already pressed
    //and to store the state
    uint8_t test = 1;

    // Code for the GPIO pins taken from [13] and
    // edited as per the requirement.
    //A GPIO configuration structure for setting the
    //Input pin for the button.
    gpio_config_t io_conf1 = {};
    //bit mask of the pins,
    io_conf1.pin_bit_mask = GPIO_INPUT_PIN_SEL;
    //set as input mode
    io_conf1.mode = GPIO_MODE_INPUT;
    //enable pull-down mode
    io_conf1.pull_down_en = 1;
    //disable pull-up mode
    io_conf1.pull_up_en = 0;
    //interrupt disable
    io_conf1.intr_type = GPIO_INTR_DISABLE;
```

```
    //configure GPIO with the given settings
    gpio_config(&io_conf1);

    gpio_config_t io_conf;
    // disable interrupt
    io_conf.intr_type = GPIO_INTR_DISABLE;
    // set as output mode
    io_conf.mode = GPIO_MODE_OUTPUT;
    // bit mask of the pins that you want to set,e.g.GPIO15
    io_conf.pin_bit_mask = GPIO_OUTPUT_PIN_SEL;
    // disable pull-down mode
    io_conf.pull_down_en = 0;
    // disable pull-up mode
    io_conf.pull_up_en = 0;
    // configure GPIO with the given settings
    gpio_config(&io_conf);
    // error variables created to store and evaluate errors.
    esp_err_t ret;
    esp_err_t retone;
    // variable to store the data from the slave device.
    uint32_t rxdata = 0;
    // array to store value of each transaction.
    uint32_t ADC_Value[300] = {0};
    // configuration for the SD card.
    // Code for SD card taken from [12].
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
#ifdef CONFIG_EXAMPLE_FORMAT_IF_MOUNT_FAILED
        .format_if_mount_failed = true,
#else
        .format_if_mount_failed = false,
#endif // EXAMPLE_FORMAT_IF_MOUNT_FAILED
        .max_files = 5,
        .allocation_unit_size = 16 * 1024};
    sdmmc_card_t *card;
    const char mount_point[] = MOUNT_POINT;
    ESP_LOGI(TAG1, "Initializing SD card");
    ESP_LOGI(TAG1, "Using SDMMC peripheral");
    sdmmc_host_t host = SDMMC_HOST_DEFAULT();
    sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
```

```
    // To use 1-line SD mode, uncomment the following line:
    slot_config.width = 1;
    // The following lines are internally pulled up
    // to satisfy the requirements of the SD card interface.
    // CMD, needed in 4- and 1- line modes
    gpio_set_pull_mode(15, GPIO_PULLUP_ONLY);
    // D0, needed in 4- and 1-line modes
    gpio_set_pull_mode(2, GPIO_PULLUP_ONLY);
    // D1, needed in 4-line mode only
    gpio_set_pull_mode(4, GPIO_PULLUP_ONLY);
    // D2, needed in 4-line mode only
    gpio_set_pull_mode(12, GPIO_PULLUP_ONLY);
    // D3, needed in 4- and 1-line modes
    gpio_set_pull_mode(13, GPIO_PULLUP_ONLY);

// The following lines check for the SD card error:
    retone = esp_vfs_fat_sdmmc_mount(mount_point, &host,
    &slot_config, &mount_config, &card);

    if (retone != ESP_OK)
    {
        if (retone == ESP_FAIL)
        {
            ESP_LOGE(TAG1, "Failed to mount filesystem. "
                           "If you want the card to be formatted,
                            set the EXAMPLE_FORMAT_IF_MOUNT_FAILED
                           menuconfig option.");
        }
        else
        {
            ESP_LOGE(TAG1, "Failed to initialize the card (%s). "
                           "Make sure SD card lines have pull-up
                           resistors in place.",
                    esp_err_to_name(retone));
        }
        return;
    }
    //Prints the SD card information.
```

```
sdmmc_card_print_info(stdout, card);
ESP_LOGI(TAG1, "Opening file");
// A new file is created.
FILE *f = fopen(MOUNT_POINT "/hello.txt", "w");
if (f == NULL)
{
    ESP_LOGE(TAG1, "Failed to open file for writing");
    return;
}
//Input the first line of the text file
fprintf(f, "Hello %s!\n", card->cid.name);
fclose(f);
ESP_LOGI(TAG1, "File written");

// spi device handle
spi_device_handle_t spi;

// bus configuration
spi_bus_config_t bus_cfg = {
    .mosi_io_num = PIN_NUM_MOSI,
    .miso_io_num = PIN_NUM_MISO,
    .sclk_io_num = PIN_NUM_CLK,
    .quadwp_io_num = -1,
    .quadhd_io_num = -1,
    .max_transfer_sz = SOC_SPI_MAXIMUM_BUFFER_SIZE,

};
// device configuration
spi_device_interface_config_t sensor = {
    .command_bits = 0,
    .address_bits = 0,
    .dummy_bits = 0,
    .mode = 0, // SPI mode 0 is used with CPOL=0 and CPHA=0.
    .duty_cycle_pos = 128,
    .cs_ena_pretrans = 0,  // 0 not used
    .cs_ena_posttrans = 0, // 0 not used
    .flags = 0,            // 0 not used
    .queue_size = 1,
    .pre_cb = NULL,
```

```
    .post_cb = NULL,
    .clock_speed_hz = CLOCK,
    .input_delay_ns = 0,
    .spics_io_num = PIN_NUM_CS};
// spi transaction struct
spi_transaction_t transaction = {
    .flags = 0,
    .cmd = 0,
    .addr = 0,
    .length = 16,
    .rxlength = 16,
    .user = NULL,
    .tx_buffer = NULL,
    .rx_buffer = &rxdata

};

// initialize the bus with DMA disabled. 0 means DMA disabled.
ret = spi_bus_initialize(HOST, &bus_cfg, 0);
// Check for error
ESP_ERROR_CHECK(ret);
// add the ADC as a device.
ret = spi_bus_add_device(HOST, &sensor, &spi);
ret = spi_device_acquire_bus(spi, portMAX_DELAY);

while (1)
{

    if (test == 1)
    {
        // check if button pressed
        pin = gpio_get_level(GPIO_NUM_22);
        if (pin == 0)
        {
            test = 2;
            //debounce delay pf 300 miliseconds
            vTaskDelay(300 / portTICK_PERIOD_MS);
        }
    }
```

```
        if (test == 2)

        {

            for (int i = 0; i < 300; i++)
            {

                gpio_set_level(GPIO_OUTPUT_IO_0, 0);
                spi_device_polling_transmit(spi, &transaction);
                ADC_Value[i] = rxdata;
                gpio_set_level(GPIO_OUTPUT_IO_0, 1);
            }

            for (int j = 0; j < 300; j++)
            {
                // reads from the lower and upper nibble
                // to combine it into a 16 bit integer
                ADC_Value[j] = SPI_SWAP_DATA_RX(ADC_Value[j], 16);
                // shifts the bits one time to the right to get
                // rid of one trailing zero as required by the ADC
                ADC_Value[j] = ADC_Value[j] >> 1;
                // gets rid of the leading leading zero
                // as required by the ADC
                ADC_Value[j] = ADC_Value[j] & 0x3FFF;
                f = fopen(MOUNT_POINT "/hello.txt", "a");
                fprintf(f, "%d\n", ADC_Value[j]);
                fclose(f);
            }
            // state changed so it checks for the push-button again
            test = 1;
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        }
    }
    esp_vfs_fat_sdcard_unmount(mount_point, card);
    ESP_LOGI(TAG1, "Card unmounted");
    spi_device_release_bus(spi);
}
```

3.2 SPI

SPI Modes		
Mode	**Clock Polarity - CPOL**	**Clock Phase - CPHA**
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

Table 3.1: SPI modes provided by the ESP32 MCU [10]

The SPI interface provided by the ESP32 is a 4-wire protocol having the clock signal, the two data lines (one that transfers data from the master to slave and the other which transfers data from the slave to the master), and one clear pin which selects with which device the ESP32 will be communicating with. [10]

The ESP-IDF SPI Master driver consists of three main components. Namely: the bus definition, the device definition and the lastly, the transmission phase. The bus definition defines the pins that are to be used for the SPI. The device definition defines, the SPI mode, the clock speed (which determines the transaction speed, the CS pin, and the bit order among other things). This gives the user the possibility to use different clock speeds for different devices connected to the same SPI bus. The transmission phase starts the SPI transaction the mode of which can be polling or interrupt based. [10]

Keeping in view of the requirements just discussed, a bus configuration structure is created. The description of the fields inside a bus structure are given below:

Bus Configuration	
Field Name	**Description**
.mosi_io_num	The MOSI pin being used (-1 as it is not used)
.miso_io_num	The MISO pin being used (e.g 19)
.sclk_io_num	The CLOCK pin being used (e.g 18)
.quadwp_io_num	Used to stop overwrite functions [1]. Only for 4 bit transactions [8]. (Not used).
.quadhd_io_num	Used to stop transactions to a specific slave [6]. Only for 4 bit transactions [8]. (Not used).
.max_transfer_sz	SOC_SPI_MAXIMUM_BUFFER_SIZE (The default value is 4092 if set to 0 and DMA is being used, otherwise the value is set to max buffer size [8].

Table 3.2: SPI Bus configuration structure [1, 6]

Device Configuration	
Field Name	**Description**
command_bits	The number of bits that are to be sent to the slave device. [8]
address_bits	The number of bits for the address that the slave is expecting. [8]
dummy_bits	The number of bits are to be inserted after the address phase but before the data phase. [8]
mode	The mode of the SPI (i.e the CPHA, CPOL) [8]
duty_cycle_pos	Used to set the duty cycle of the clock. Can be set between 1 and 256. 128 is the default value corresponding to a duty cycle of 50%. [8]
cs_ena_pretrans	Used to control the activation timing of the CS signal before the transmission of data. Can be set from 0 to 16 corresponding to the number of clock cycles. [8]
cs_ena_posttrans	Used to control the deactivation timing of the CS signal after data transmission. Can be set from 0 to 16 corresponding to the number of clock cycles.[8]
flags	Stores the bitwise OR value of the five SPI_TRANS_* flags. [10] [8]
queue_size	Sets the number of transaction that can be in the process at one time. [8]
pre_cb	An interrupt based callback function to be called before the start of the trancsaction.[8]
post_cb	A interrupt based callback function to be called after the end of the transaction.[8]
clock_speed_hz	Clock speed of the SPI can only be divisors of 80MHz.[8]
input_delay_ns	Delay in nano seconds before the MISO data is ready on the line. [8]
spics_io_num	Pin number of the CS line of the slave. Set to -1 if CS is to be user controlled. [8]

Table 3.3: Slave device configuration structure [8]

Transaction Configuration	
Field Name	**Description**
flags	Contains the bitwise OR value of the transaction flags. (uint32_t) [8]
cmd	The value sent to the device during the command phase. (uint16_t) [8]
addr	The value sent to the slave device during the address phase. (uint64_t) [8]
length	Overall length of the data. [8]
rxlength	Length of the data to be received by the ESP32. [8]
user	Can be used by the user to store arbitrary values. [8]
tx_buffer	Stores the address of the variable which contrains the data to be sent to the slave. [8]
rx_buffer	Stores the address of the variable which stores the data sent by the slave device to the ESP32. [8]

Table 3.4: SPI transaction configuration structure [8]

After the aforementioned structures are set with the required values, a SPI handle is created with the keyword spi_device_handle_t. Then the spi_bus_initialize() function is called with the parameters: SPI host which can be HSPI or VSPI, the address of the bus configuration structure and the one of the two DMA channels or no DMA. Then the slave device should be associated with the bus using the spi_bus_add_device() function. Its parameters are host, address of the device configuration structure, and the address of the device handle. Thereafter the spi_device_acquire_bus() function is called since there is only one SPI slave which is needed to be communicated with. Its parameters are the spi device handle and portMAX_DELAY which cannot be changed for the time being. Interrupt based or polling transaction can now be called using spi_device_polling_transmit for polling or spi_device_transmit() interrupt based transmissions. The parameters are the device handle and address of the transaction structure. [8]

3.3 SD Card

The code for the SD card is taken from [12]. First, the esp_vfs_fat_sdmmc_mount_config_t structure is set, specifying if the card should be formatted if mount was failed, the maximum number of files, and the allocation size. Thereafter a sdmmc_card_t pointer variable is created. A constant char array is also created and equated to /sdcard. The SDMMC_HOST_DEFAULT() function is called with its return value saved into the host variable and the SDMMC_SLOT_CONFIG_DEFAULT() function is also called its value is saved into an sdmmc_slot_config_t variable. slot_config.width = 1 is also written so only one width data line mode is used due to the strapping pin requirements for the 4-bit line width mode. The SD card is mounted using the esp_vfs_fat_sdmmc_mount function the parameters of which are /sdcard, address of the host, address of the slot_config variable, address of the mount_configuration variable and the address of the card variable. Different errors can be checked by saving the return value of the function into an error variable and comparing that value with keywords such as ESP_FAIL. The information of the SD card, once mounted, can be output on the console through the function sdmmc_card_print_info(stdout, card).

Data can be saved into the SD card using file functions provided by C.

3.4 GPIO pins configuration for the push button and the CS signal

Code for the GPIO was taken from [13] and has been edited as per the requirement. GPIO pins have to be configured before they can be utilized as an input or an output. The push-button pin has to be configured as an input pin so that is can be used to sense the state of being pressed or released. The CS pin was also configured as an output pin so it could be user controlled. These functionalities were done using a gpio_config_t structure.

The fields that have to be filled are given in the following table:

GPIO Configuration	
Field Name	**Description**
pin_bit_mask	Contains the information of the pin number. The number 1 should be bitwise shifted to the left x number of bits. Here x is the pin number.
mode	The mode of the pin either input or output.
pull_down_en	Should be set to 1 if internal pull down is required.
pull_up_en	Should be set to 1 if internal pull-up is required.
intr_type	Interrupt to be activated on a particular edge or no interrupt, if not required.

Table 3.5: GPIO configuration structure [7]

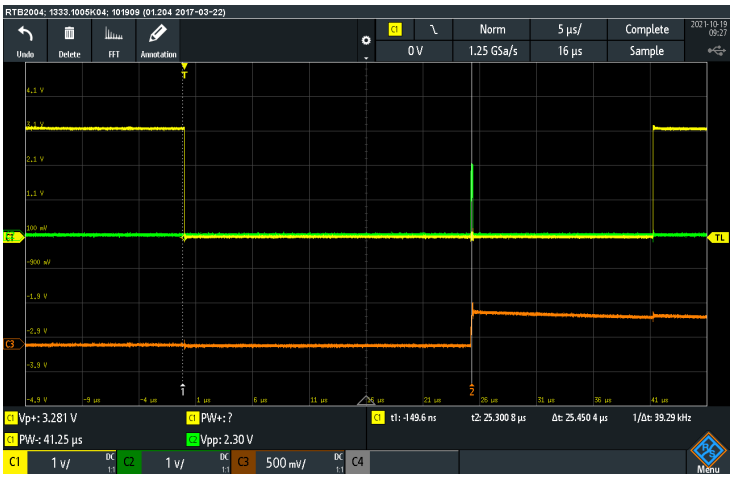
Once the values are set for the particular configuration structure, the address of the structure should be passed into the gpio_config function. This would result in the GPIO pins configured for use. [7]

3.5 Timings Considerations

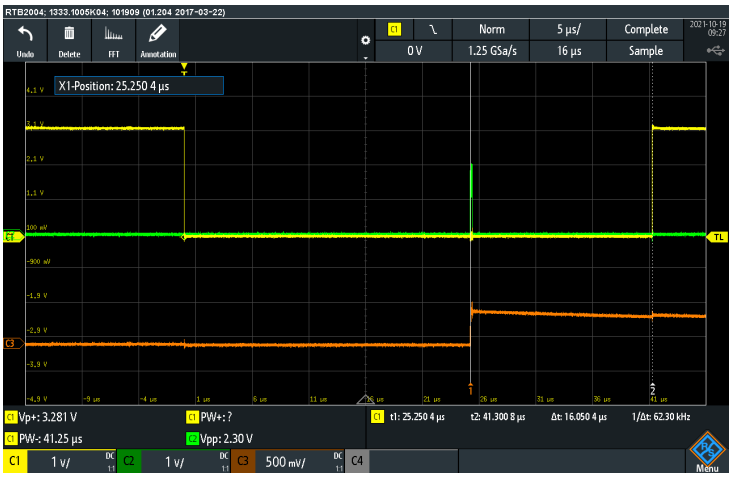
In this section, timing diagrams of the various methods required to control the clear signal will be analysed. As mentioned before, there exists two ways to control the clear signal. The following screenshot are taken from the oscilloscope. The yellow line represents the CS signal, the orange line represents the incoming data on the MOSI pin and the green line represents the SPI clock.

3.5.1 User controlled clear signal with the GPIO pins

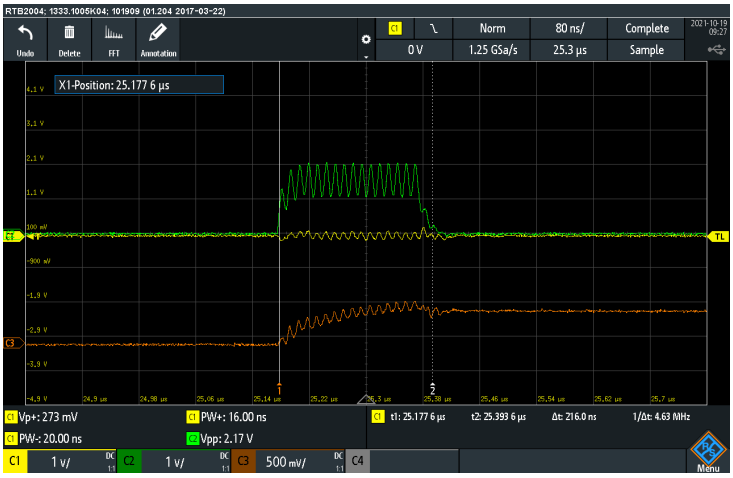
Interrupt based single transactions



(a) 25.45 µs delay.



(b) 16 µs delay.



(c) data read time 216.0 ns.

Figure 3.1: Duration of a single transaction in an interrupt based transmission using a GPIO based clear signal

Interrupt based multiple transactions with a for-loop

(a) 25.7 µs delay between each transaction.

(b) 13.25 µs delay between clear signal and the start of transaction.

(c) 11.95 µs delay between the end of transaction and the clear signal.

Figure 3.2: Duration of multiple transactions (for-loop) in interrupt based transmissions using a GPIO based clear signal

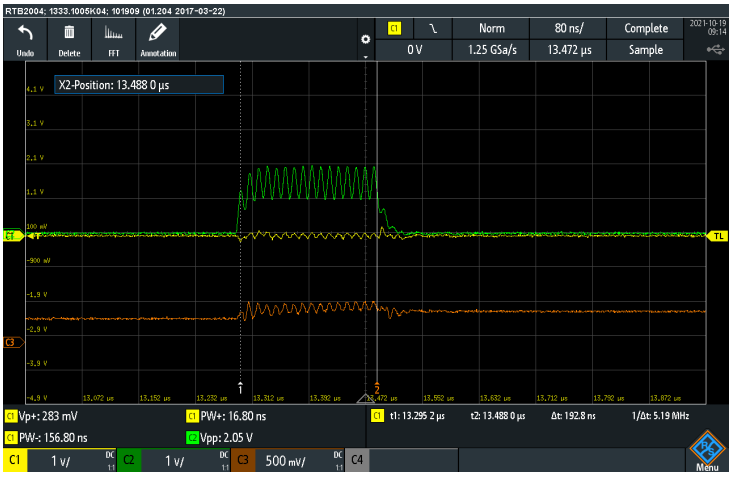
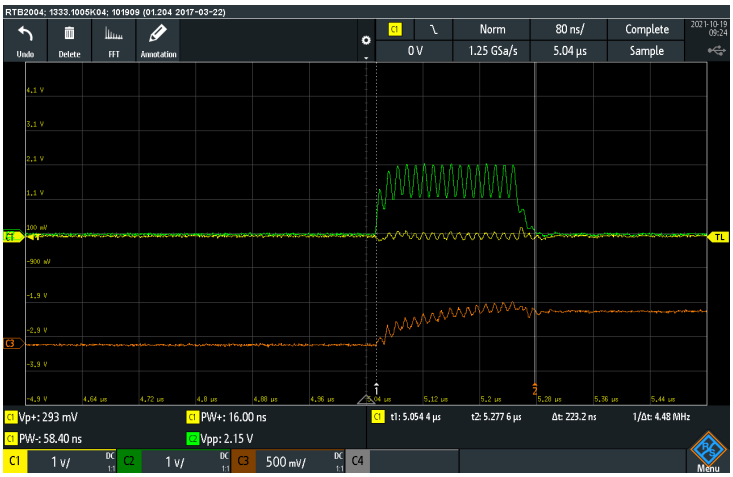


Figure 3.3: Microscopic view of a single transaction in figure 3.2. Read time of 192.8 ns.

Polling based single transactions

(a) 5.06 µs delay between the CS and start of the transaction.

(b) 1.33 µs delay between the CS and end of the transaction.



(c) Read time of 223.2 ns

Figure 3.4: Duration of a single polling transaction using a GPIO based clear signal

Polling based multiple transactions

(a) Delay of 6.64 µs between transactions.

(b) 5.04 µs delay between the CS and start of the transaction.

(c) 1.28 µs delay between the CS and end of the transaction.

Figure 3.5: Duration of a multiple polling transactions using a GPIO based clear signal

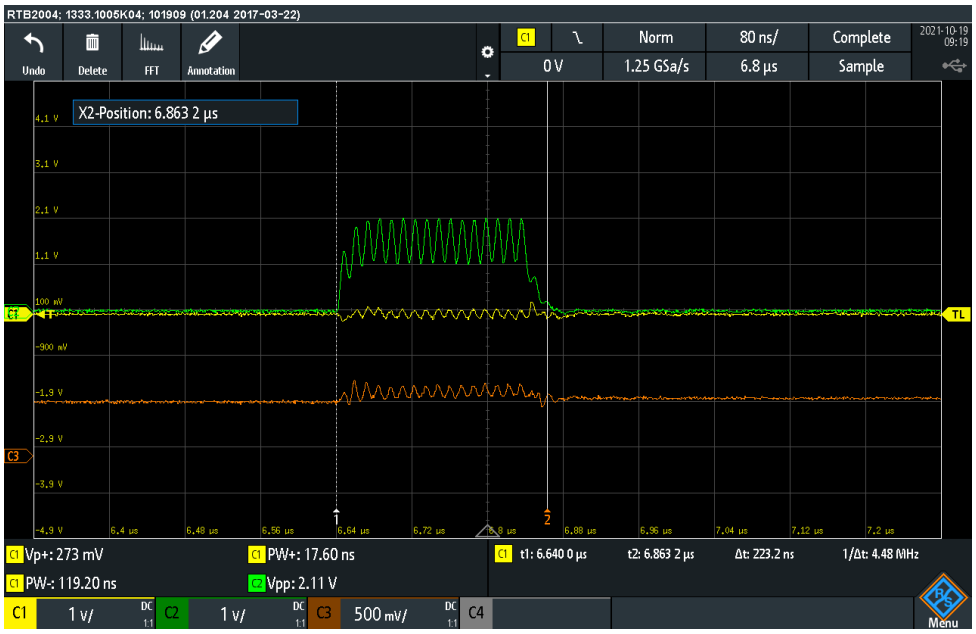


Figure 3.6: Microscopic view of a single transaction in figure 3.5. Transaction time of 223.2 ns.

3.5.2 ESP IDF SPI driver provided clear signal

Interrupt based single transactions

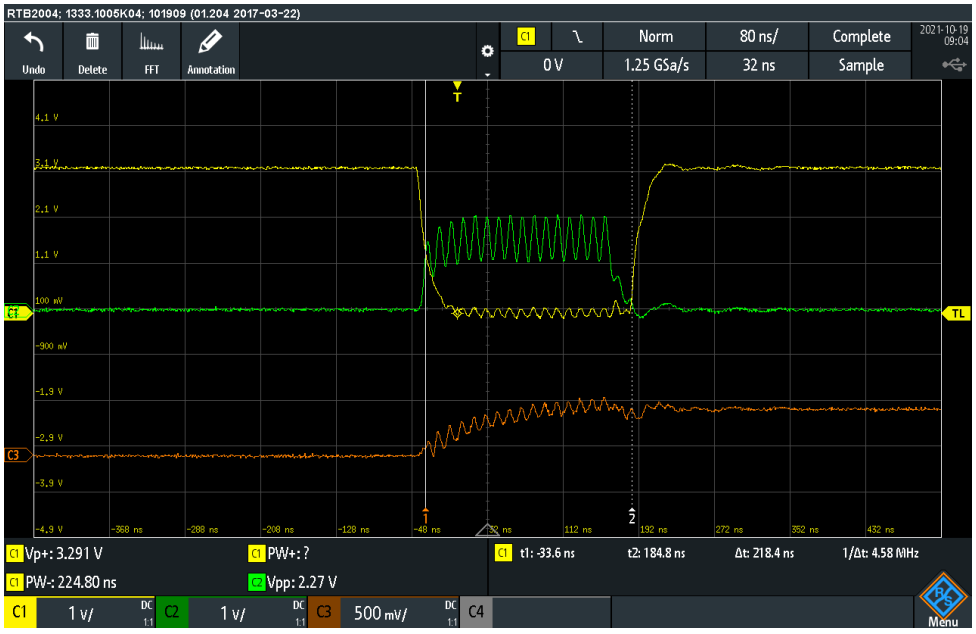
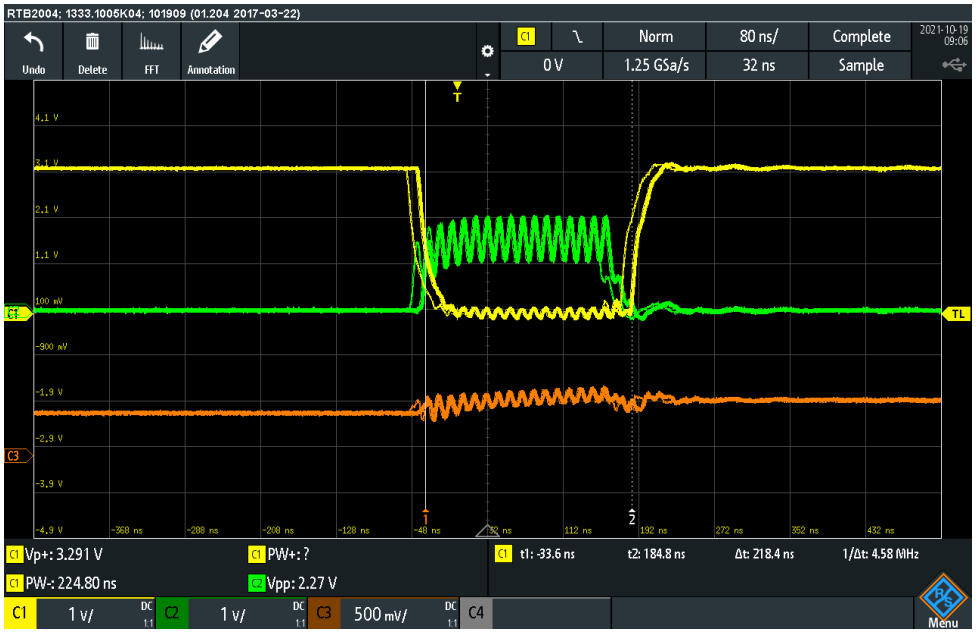
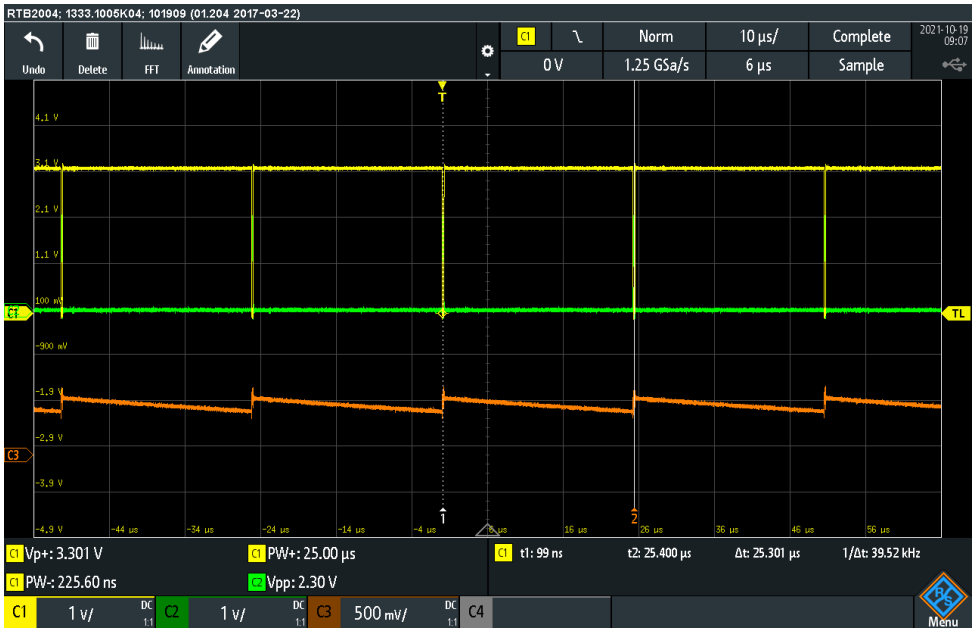


Figure 3.7: Interrupt based single transactions. Data read time of 218.4 ns.

Interrupt based multiple transactions



(a) Data read time of 218.4 ns.



(b) Delay of 25.3µs between transactions.

Figure 3.8: Interrupt based multiple transactions

Polling based single transaction

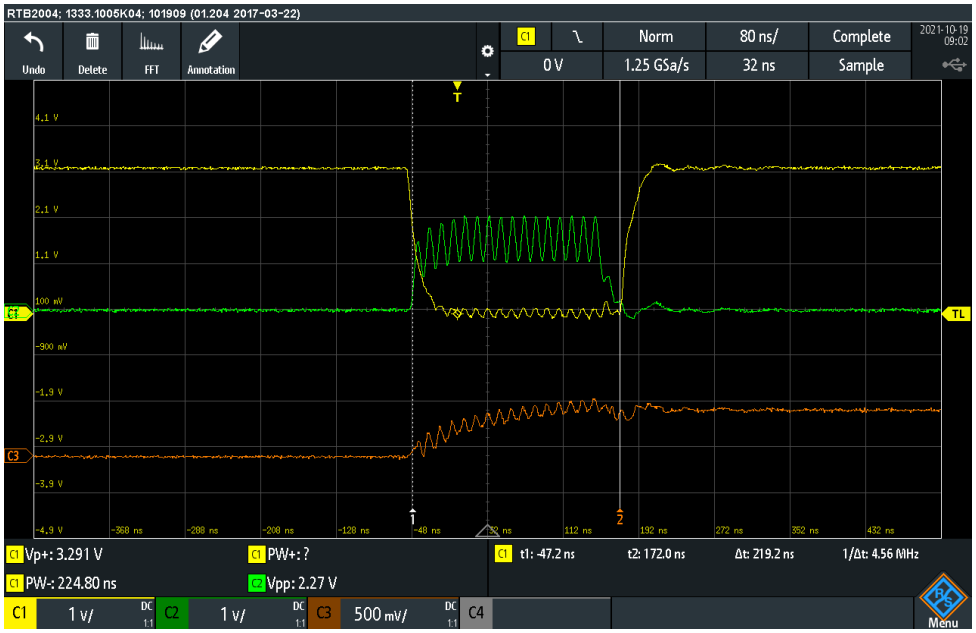
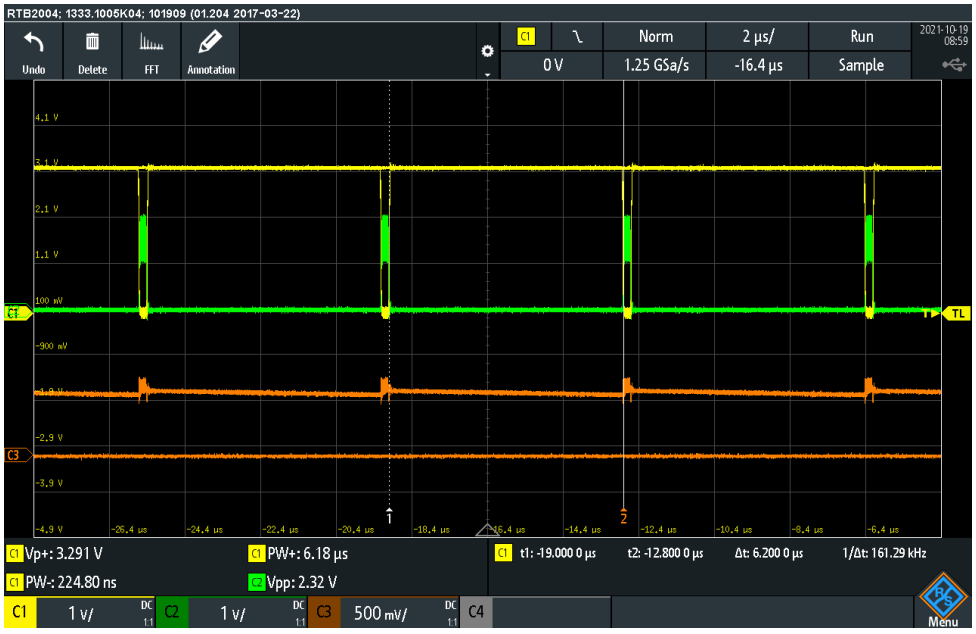
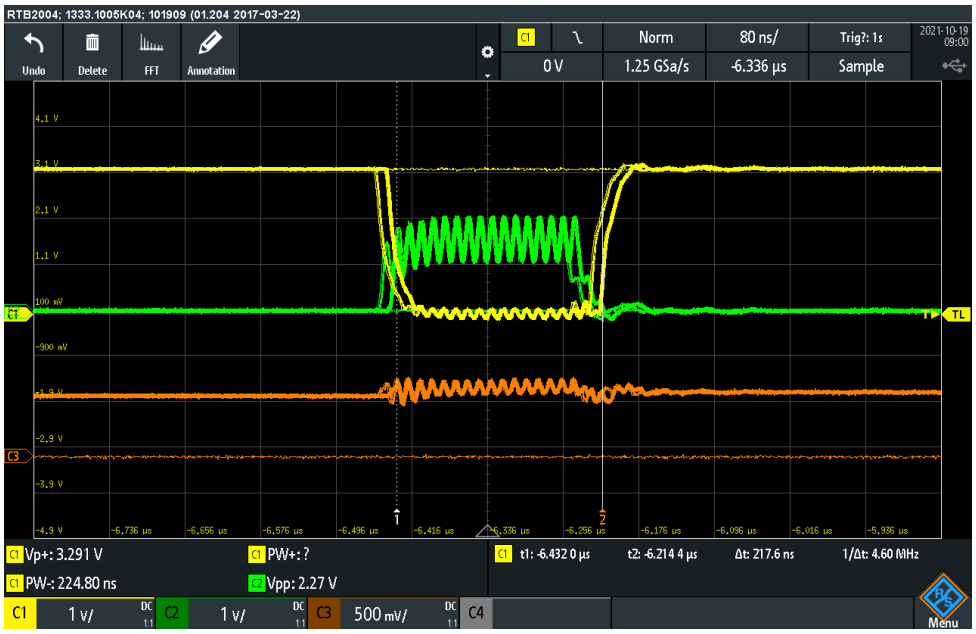


Figure 3.9: Polling based single transaction. Data read time of 219.2 ns.

Polling based multiple transactions



(a) Delay of 6.2µs between transactions.



(b) Data read time of 217.6.

Figure 3.10: Polling based multiple transactions

3.5.3 Comments

As can be seen in figure 3.1, the GPIO based clear signal introduces additional delay between the data reading and the start and end of the clock signal. When the same is implemented in a for-loop for multiple transactions, the transaction time remains almost constant for an individual transaction as seen in figure 3.3, but the transaction delay between each transaction is reduced to 25.7 µs as can be seen in 3.2.

The delay between the CS signal and the clock is significantly reduced when using a polling based method as can be seen in figure 3.4 for a single transaction. In figure 3.5, the reduced time between multiple transactions can also be seen. However, figure 3.6 again shows that the transaction time is around the same.

For the CS signal provided by the ESP-IDF drivers, there is virtually no delay between the start and end of the CS signal and the SPI clock as seen in figure 3.7, but the delay between each transaction is again around 25.3 µ as can be seen in figure 3.8 for a interrupt based method. For a polling based method, although the data read time remains about the same as can be seen in figure 3.9, the delay between each transaction is around 6.2 µ as can be seen in figure 3.10 which is around the same as if a GPIO based CS signal was provided.

3.6 Signal reconstruction

3.6.1 Working code

MATLAB was used to plot and reconstruct a sine-wave that was fed to the ADC to check for accuracy. The following code was provided by [5]:

```
x= dlmread('HELLO.txt');
v= nonzeros(x');
plot(v);
title('Reconstructed 1KHz Signal');
xlabel('Data points')
ylabel('Scaled value by the ADC');
```

The data saved into the SD card was retrieved into a PC and non-numeric characters were manually removed from the SD card. A reconstructed sine-wave of 1KHz frequency can be seen in the figure 3.11. For this conversion the polling method with a for-loop was used. Each delay between subsequent transactions was 6.2 µs as can be seen in figure 3.10a. For reference the figure 3.12 shows the sampling points of a 20KHz sine-wave.

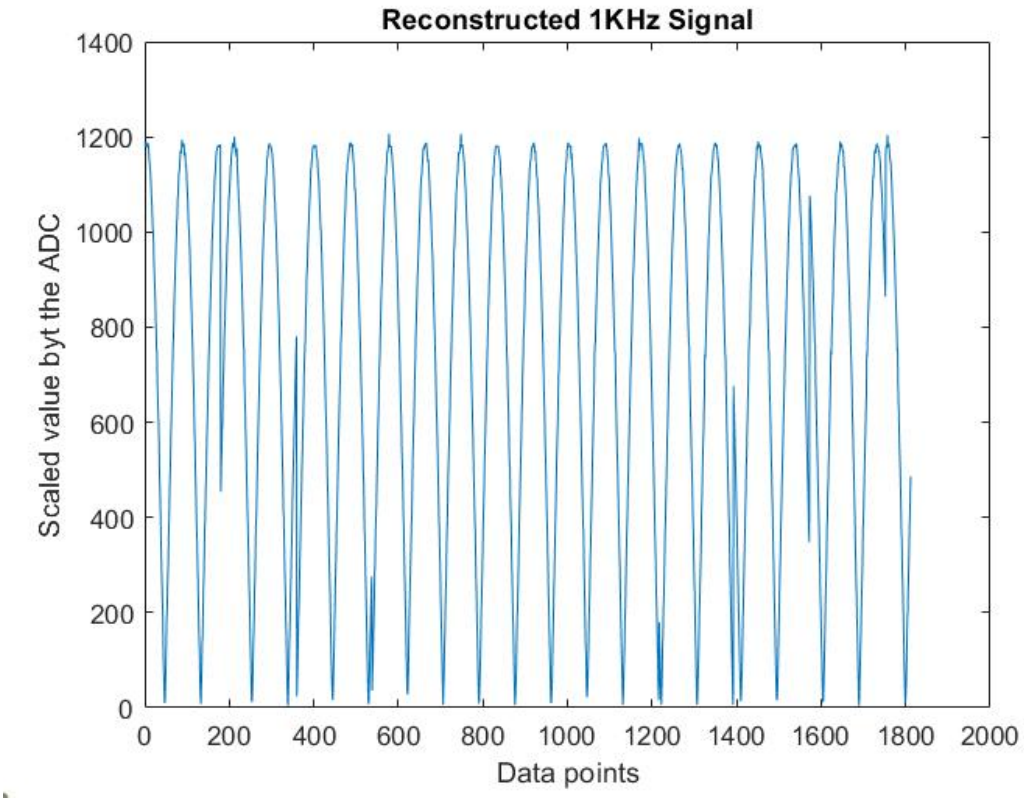


Figure 3.11: Signal reconstruction of a 1 KHz Sine-wave

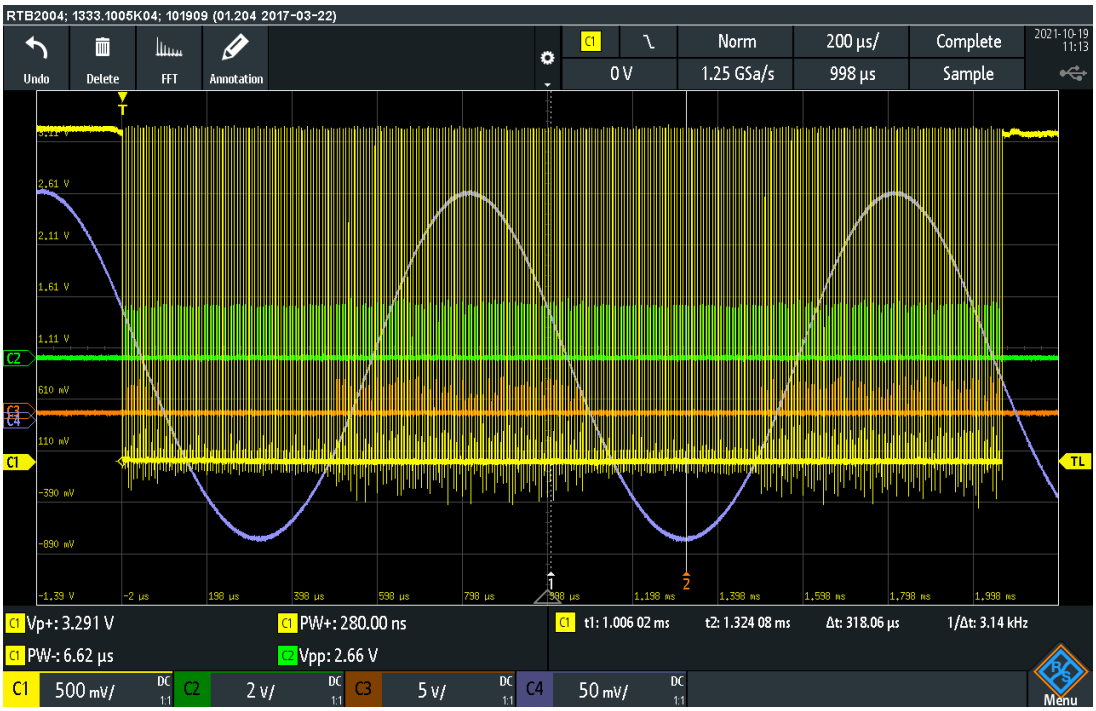


Figure 3.12: Data sampling of a 20KHz Sine-wave. 300 Sampling points are shown.

4 Summary

The conclusion of this project resulted in the understanding that the setting up time of the ESP 32 drivers for SPI transactions are too long for the application at hand. The interrupt based transactions take around 25µs while polling based transactions take 6.2µs. This would require the editing of the SPI drivers themselves (offered by the ESP IDF) or essentially, the writing of a particular SPI driver for the ESP32 MCU, which would attempt to reduce the delay between consecutive SPI transactions to a few nanoseconds. If the former is not achievable, other hardware such as an FPGA must be considered.

Bibliography

[1] ataradov. *What is the purpose of WP pin in a SPI NOR flash*. Feb. 11, 2021. URL: https://www.eevblog.com/forum/projects/what-is-the-purpose-of-wp-pin-in-a-spi-nor-flash/ (visited on 10/17/2017).

[2] BEHRTECH. *3 Advantages of IoT for Tank Monitoring in Process Industries*. 2020. URL: https://behrtech.com/blog/3-advantages-of-iot-for-tank-monitoring-in-process-industries (visited on 10/17/2021).

[3] Linear Technology Corporation. *LTC2314-14*. 2013.

[4] Piyu Dhaker. *Introduction to SPI Interface*. Analog Dialogue. 2018. URL: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html (visited on 10/17/2021).

[5] Falco Edner. Institute for Automation Technology, Chair of Measurement Technology, Faculty of Electrical Engineering and Information Technology , Otto-von-Guericke Univeristy, 2021.

[6] Ltd Espressif Systems (Shanghai) Co. *Get Started*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/ (visited on 10/26/2021).

[7] Ltd Espressif Systems (Shanghai) Co. *GPIO & RTC GPIO*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html (visited on 10/27/2021).

[8] Ltd Espressif Systems (Shanghai) Co. *SPI Master Driver*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_master.html#spi-master-driver (visited on 10/21/2021).

[9] Nicholas Humfrey. *Arduino Library List*. 2021. URL: https://www.arduinolibraries.info/architectures/esp32 (visited on 10/17/2021).

[10] Neil Kolban. *Kolban's Book on ESP32*. https://leanpub.com/kolban-ESP32(visited 2021-10-21). leanpub, 2018.

[11] Espressif Systems (Shanghai) Co. Ltd. *ESP-IDF Programming Guide*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/ (visited on 10/17/2021).

[12] Martin Vychodil. *SPIbus*. 2020. URL: https://github.com/espressif/esp-idf/blob/bcbef9a8db54d2deef83402f6e4403ccf298803a/examples/storage/sd_card/sdmmc/main/sd_card_example_main.c.

[13] Michael, suda-morris, and Angus Gratton. *generic_gpio*. 2021. URL: https://github.com/espressif/esp-idf/blob/bcbef9a8db54d2deef83402f6e4403ccf298803a/examples/peripherals/gpio/generic_gpio/main/gpio_example_main.c.

[14] Arduino Modules. *KY-004 KEY SWITCH MODULE*. 2020. URL: https://arduinomodules.info/ky-004-key-switch-module/ (visited on 10/17/2021).

[15] Natanael Josue Rabello. *SPIbus*. 2017. URL: https://github.com/natanaeljr/esp32-SPIbus.

[16] Steffen Ronalter. *3 Advantages of IoT for Tank Monitoring in Process Industries*. DEARDEVICES. 2020. URL: https://deardevices.com/2020/05/31/spi-cpol-cpha/ (visited on 10/17/2021).

[17] ESPRESSIF SYSTEMS. *ESP32-DevKitC*. 2021. URL: https://www.espressif.com/en/products/devkits/esp32-devkitc (visited on 10/17/2021).

[18] ESPRESSIF SYSTEMS. *ESP32-WROOM-32 Datasheet*. Version 3.2. 2021.

[19] VEGA. *Level and pressure instrumentation for the chemical industry*. URL: https://www.vega.com/-/media/pdf-files/industry-brochures/28286-en-level-and-pressure-measurement-technology-for-the-chemical-industry.pdf (visited on 10/17/2021).