# SPI Logic Analyser

## Contents

# Introduction

I always wanted a logic analyser, but they are very expensive for hobby use. So I decided to build one for my own.

Although I knew I would never reach the capabilities of a 800$ or more device, this way I can learn a lot about programming and data acquisition.

# Prerequisites

## Hardware

- Raspberry Pi Zero W
- Attiny44v
- Jumper Cables

## Software

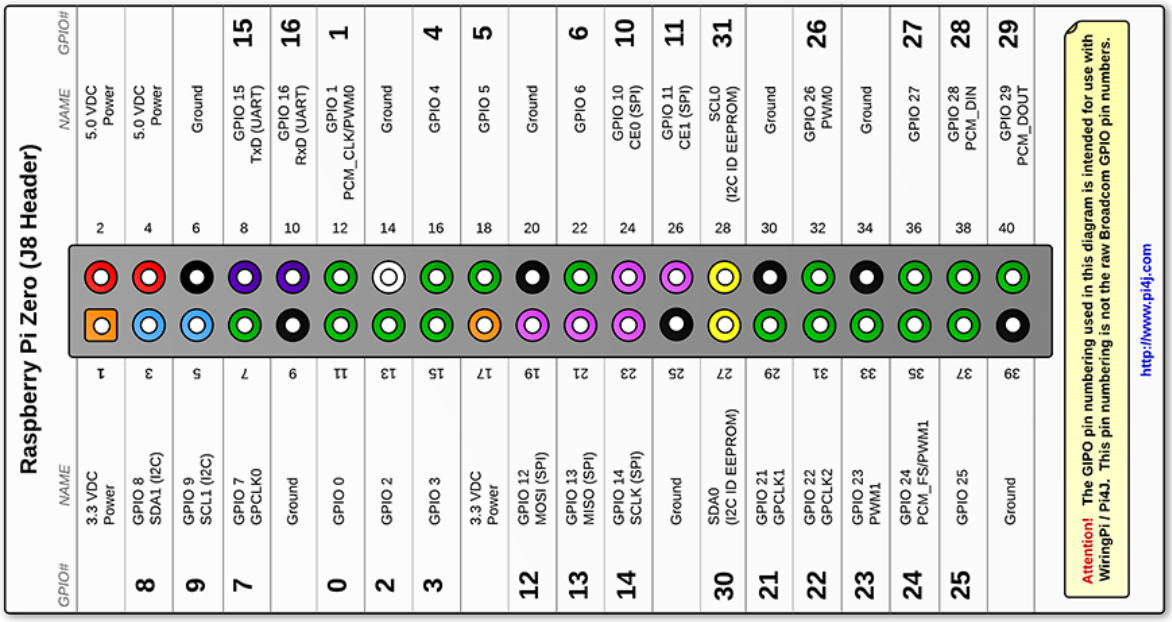
- Python 3
- Matplotlib
- WiringPi for Python

Note that SPI has to be enabled in the Raspi-Config.

# Description

I will mainly focus on software, since all hardware is either in the Attiny_von_Hannes project or consists exclusively of jumper wires.

The only thing to point out is the GPIO pins used on the Raspberry which are:
17: 3.3V
20: Ground
19: MOSI
21: MISO
23: SCLK




The connection to the Attiny is 1:1 so MOSI ⇔ MOSI etc.

Also note, that when using the Attiny as master (which here is not the case) the data will still get shifted out at the DO pin, which is MISO.

```
.DEVICE attiny44                  ; device to check memory
.INCLUDE "./include/attiny44.asm" ; register map made by me

Init:
      ldi r16, high(RAMEND) ; load high part of adress
      out SPH, r16 ; set stack pointer to top of RAM
      ldi r16, low(RAMEND) ; load low part of adress
      out SPL, r16 ; set stack pointer to top of RAM
      ldi r16, (1<<PORTA5) ; miso and clock are outputs
      out DDRA, r16 ; data directions are set
SlaveSPIInit:
      ldi r16, (1<<USIWM0)|(1<<USICS1)
      out USICR, r16
      ldi r18, (1<<USIOIF) ; set flag bit

Main:
Sample:
      in r16, PINA ; read Port A
      andi r16, 0b10001111 ; Pin 0, 1, 2, 3, 7
      swap r16 ; swap nibbles in r18
      in r17, PINB ; read Port B
      andi r17, 0b00000111 ; Pin 0, 1, 2
      or r16, r17 ; combine the registers
LoadTransferRegister:
      out USIDR, r16 ; put data into transmission data register
      out USISR, r18 ; clear flag
SlaveSPITransfer_loop:
      in r16, USISR ; load usi status register
      sbrs r16, USIOIF ; if counter is overflow, data is finished
sending
      rjmp SlaveSPITransfer_loop ; otherwise, check again
ReturnValue:
      rjmp Main ; r16 contains read data
```

The code for the Attiny is very short, which is important to maximize speed.

In the initialization, first basics are handled. This means that the stack pointer is set to the end of the useable RAM address.
The most important thing here is to set (only) the DO pin to an output using its data direction register. Without this, the SPI interface of the ATtiny could not send anything.

Next is the initialization of the SPI interface. More correct would be to say the initialization of the USI interface to a three-wire mode, compatible to SPI. USIWM0 sets exactly this mode. The USICS sets the clock source to an external clock, so the ATtiny functions as a slave device and lets the Raspberry handle the data and timing.

The main loop starts with sampling the inputs. To do so, both ports A and B are read. It could be optimized by reading them directly after one another to make the data even more synced but for my needs this is enough for now.
I read both ports (reading is using the pin address for “port in”) because from port B I can not use Bit 3 since it is reset and from port A I can not use the Bits corresponding to the SPI interface and the sum of the remaining Bits is exactly 8 so this is a nice fit.
What I just described is the reasoning behind the masking of both registers using the andi instruction.
The easiest way to combine both registers is to swap the nibbles of one of the registers. This is faster then shifting the whole register since for each shift one clock cycle would be used and this way the swap only takes one clock cycle.
The or instruction combines the registers into one 8 Bit value.

Now the sampled values are copied into the transfer data register.
The interrupt flag for counter overflow of the USI interface is cleared by writing a 1 to it. Since I want to avoid unnecessary clock cycles I stored the Byte to do so once in the initialization so I don’t have to do it every cycle of the main loop. There are enough working registers left to do so.

In the transfer loop all I do is to check the counter overflow bit. Therefore I read the status register and then check it. If the bit is not set, I check again, otherwise I restart the main loop.
The read value during the SPI data transfer does not matter so I can simply ignore it.
The overflow flag is set when the counter counted enough clock edges. This means that external clock edges on the SPI clock pin increase the value. When all 8 Bits are transferred, the flag gets set.

```
import wiringpi
import time
import matplotlib.pyplot as plt

wiringpi.wiringPiSetupGpio()  # For GPIO pin numbering
channel = 0
speed = 1_000_000
wiringpi.wiringPiSPISetup(channel, speed)
buf = bytes([0x00])

masks = [0x01, 0x02, 0x04, 0x08,
      0x10, 0x20, 0x40, 0x80]

sample_time = 10
values = [[0, 0, 0, 0, 0, 0, 0, 0]]
timestamps = [time.time()]
target_time = timestamps[-1] + sample_time

while timestamps[-1] < target_time:
      retlen, retdata = wiringpi.wiringPiSPIDataRW(0, buf)
      values.append([int(retdata[0] & mask != 0) for mask in masks])
      timestamps.append(time.time())

if len(timestamps) != len(values):
      print("ERROR: Sample values and timestamps don't match!")
      quit()

print(f"Sampled values: {len(values)}")

bitwise = []
for bit in range(8):
      bitwise.append([n[bit] for n in values])

fig, axes = plt.subplots(8, 1, sharex=True)

for axis in range(8):
      axes[axis].plot(timestamps, bitwise[axis])
      axes[axis].set_ylim(-0.5, 1.5)
      axes[axis].set_xlabel("Timestamp")
      axes[axis].set_ylabel(f"Bit {axis}")

plt.show()
```

The python code first initializes the SPI interface of the Raspberry using the WiringPi library for Python.
I set the bitmasks to later decode the incoming values.
The sample_time defines how long samples are taken. During this time data is sampled as fast as possible. To compensate for differences in sample time, timestamps are collected.

The sampling is done in a while loop, running until the target time is reached.
I transmit a buffer containing 0x00 because each read also needs a write with SPI. The incoming data is then checked using the bitmask. This could be done after the sampling to increase efficiency but it seems to me that the bottleneck is the ATtiny already so this is not necessary for now.

If the number of sample values does not match the number of timesteps the program stops since there was a sampling error and the resulting times can not be guaranteed to be right.

In the next step I separate the sampled arrays into each Bit individually. This makes plotting way easier.

To plot I create 8 subplots, one for each Bit. They all share the same X-Axis since the sampling times are synced, except for the 3 or 4 clock cycles I described above.

I decided to go for it and try to reach more speed. To do so, I first moved the bit masking out of the capture loop, as I explained before. This way I managed to double the sampling rate, so it is clear, that the bottleneck is the Python code.

So I had no choice but to program at least the sampling part in C. At first I ran into a lot of problems since the last time I coded in C is quite a while ago, but after reading the right how-tos it went more or less smoothly.
There was a problem where I got an unexplainable “Segmentation fault” error when trying to write the captured data to a file but other than that no major difficulties.

I had to install an FTP server on the Raspberry to inspect the captured data since the file was too big to do so on the Raspberry itself.

Install Pure-FTPd:
```
sudo apt-get install pure-ftpd
```

Create a new user group and a new user:
```
sudo groupadd ftpgroup
sudo useradd ftpuser -g ftpgroup -s /sbin/nologin -d /dev/null
```

Make a new directory named FTP for the first user:
```
sudo mkdir /home/pi/FTP
```

Make sure the directory is accessible for ftpuser :
```
sudo chown -R ftpuser:ftpgroup /home/pi/FTP
```

Create a virtual user named upload, set home directory /home/pi/FTP, add password:
```
sudo pure-pw useradd upload -u ftpuser -g ftpgroup -d /home/pi/FTP
-m
```

A password of that virtual user will be required after this command line is entered.

```
sudo pure-pw mkdb
```

The number 60 is only for demonstration, make it as small as necessary (best case 0):
```
sudo ln -s /etc/pure-ftpd/conf/PureDB /etc/pure-ftpd/auth/60puredb
```

Restart the program:
```
sudo service pure-ftpd restart
```

Test it with an FTP client, like FileZilla.
For me: user is “upload” and password is “admin”.

```
#include <wiringPiSPI.h> // when building link "-l wiringPi"
#include <stdio.h>
#include <time.h> // when building link "-l rt"
#include <stdlib.h>

#define MAX_TIME 10

FILE *fp;

int main()
{
      int channel = 0;
      int speed = 500000;
      wiringPiSPISetup (channel, speed);
      int len = 8;
      unsigned long int this_time;
      unsigned long int start_time;
      struct timespec gettime_now;
      fp = fopen("/tmp/test.txt", "w");

      if (fp) { // file created successfully
      clock_gettime(CLOCK_REALTIME, &gettime_now);
      start_time = gettime_now.tv_sec;
      while (1) {
            unsigned char buf = 0;
            wiringPiSPIDataRW(channel, &buf, len);
            clock_gettime(CLOCK_REALTIME, &gettime_now);
            this_time = gettime_now.tv_nsec;
            fprintf(fp, "%ld %d\n", this_time, buf);

            if (gettime_now.tv_sec - start_time >= MAX_TIME) {
                  break;
            }
      }

      fclose(fp);
      printf("Sampling done.\n");
      system("sudo cp /tmp/test.txt /home/pi/FTP");
      } else {
      printf("ERROR: Creating file failed.\nMake sure the directory
/tmp/ exists.\n");
      }
      return 0;
}
```

First a file pointer is created.

The main again consists of setup, loop and cleaning up.
First the SPI interface is set up. Channel 0 and 500kHz.
For the timing (I want x seconds captured), two unsigned long integer variables are used. One is for nanoseconds while the other is for seconds.

When the file was successfully created, the main loop starts. It is an infinite loop, exiting when the time has reached the desired length.
A one byte buffer is created and passed as a pointer to the SPI function. This is necessary, because it's value is overwritten by the incoming data. Therefore, when the transfer is complete, the buffer's value can be saved with the timestamp.

After closing the file it is copied to the FTP folder to be ready for external investigation.

When compiling, some libraries have to be linked using -l. The output file can be names using -o. Compile the file with:

```
gcc SPI_Logic_Analyser.c -l wiringpi -o logica
```

Execute it with:

```
sudo ./logica
```

However, even this way, I was not able to reach more than 5kHz.
I think the reason is, that writing the file within the loop is not fast enough, but even when writing to an array and storing the values in RAM and writing to the file afterwards I could not manage to reach higher speeds.

The last idea I had was sending a larger buffer, than just one byte. This way, I was able to go up to 100kHz, but this rate can only be used for 4000 bytes since there is a buffer size limitation on the Raspberry.

```
num_of_samples = int((sample_time * speed) / 8)
num_of_sets = int(num_of_samples / 4000) + 1
buf = bytes([0x00 for n in range(4000)])
print(f"Expecting {num_of_samples} samples in {num_of_sets} sets.")

retdatas = []

for n in range(num_of_sets):
      retlen, retdata = wiringpi.wiringPiSPIDataRW(0, buf)
      retdatas.append(retdata)
```

When repeating the measurement this way, there is a larger time gap every 4000 samples and therefore not really useful. Except of course for quick, well timed samples of 40ms.

This buffer can be increased by writing to the file “/boot/cmdline.txt”:
`spidev.bufsiz=65536`

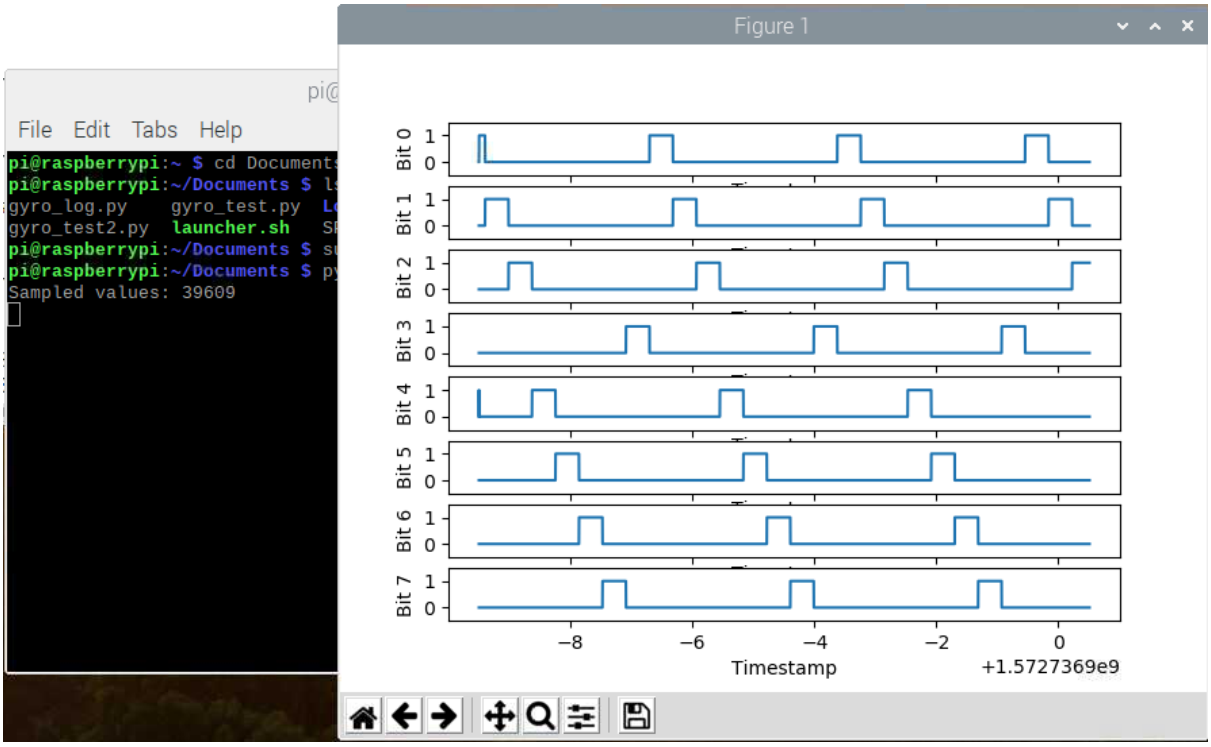
The number is the buffer size in bytes.
Keep in mind, that this file is one line only. Insert this part in front of “rootwait” and reboot.

# Results summarized

I managed to build a Logic Analyser with 8 Bit simultaneous sampling. Data rates are up to 4kHz and can be increased when using an external clock or a faster microcontroller in general.

When trying to maximize for speed I noticed that sampling with Python turned out to be the bottleneck. I managed to get up to about 8kHz with Python and then switched to C for sampling and Python for evaluation. That did not help. However, when limiting myself to 4000 samples I can get up to 100kHz, limited by the buffer size on the Raspberry Pi.

At some point the Raspberry should no longer do the calculations and only function as a data collection device however, since there is a lot of data being generated.



# Links to Sources and more Information

- Wiring Pi: http://wiringpi.com/
- FTP setup: https://www.raspberrypi.org/documentation/
-