Atmel 328-Processor for RaspberryPi

AD-Converter, Frequency measurement, Eventcounter, IR-Control

Gerhard Hepp, März 2014

Content

Overview

To connect analog signals as from a potentiometer, a temperature sensor or distance sensors there are AD-Converters ADC needed.
Unfortunately there is no great choice on add-on boards for Raspberry Pi with ADCs. One option is to use an Atmega328 controller on a breadboard for this purpose.

Sample code for Raspberry Pi is provided in python. Integration to scratch is achieved with the scratchClient-Software from heppg.de.

The 328-controller has many build in functions as Timers, SPI, ADC and more. This controller is quite popular as the arduino boards use this device.
The ADC provides 10 bit resolution, and cost for the device are a few euro and comparable to dedicated ADC devices.
The firmware provided allows for additional functionality as event counting, frequency measurement, IRC-connection.

The firmware needed can be programmed from the RPi, and the device can be easily implemented on a breadboard.

Disadvantage is the lengthy installation procedure. But with simple steps and verification points this is manageable.

The firmware for the controller is ready to use in the samples, no coding required in this area. As programming the firmware into the Atmel328 is done with the RPi, no extra programming device is needed.
The programming of the atmel device is no topic of this article.

Prerequisite is that the contoller is using the internal 8MHz RC oscillator. This is default for factory new devices.
If you get a preprogrammed device with e.g. a bootloader for arduino, the internal oscillator can be switched off. In this case you need to use a programming device, or attach an external clock signal. In the appendix, the procedure is described for this.

Setup

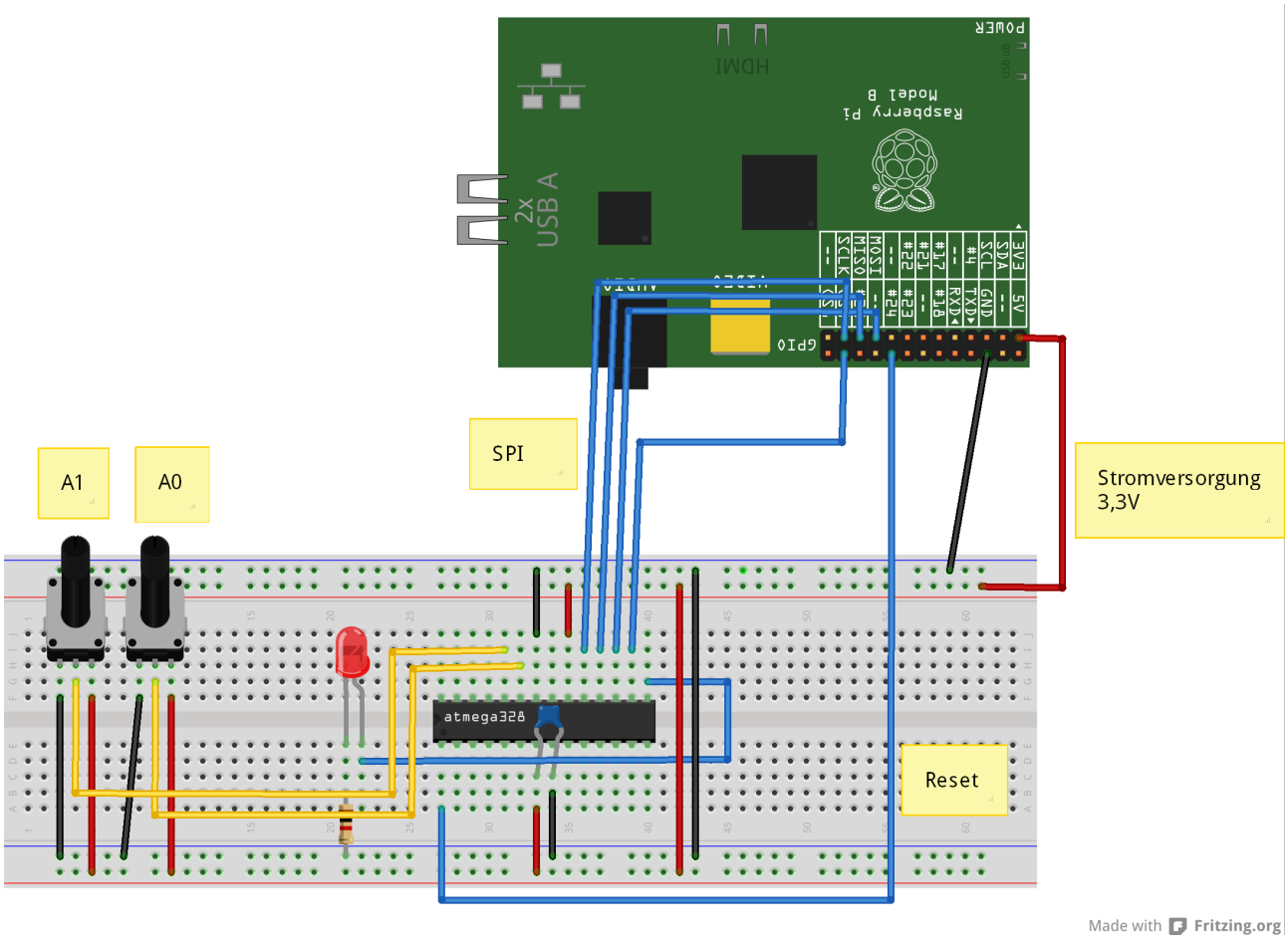
Parts list

- ATmega 328 P-PU (DIL-package)
- breadboard
- (optional) precision socket 28 pol, DIL 28 pol, 0.3 large. The socket is needed to protect the pins of the processor from bending. Set the socket into the breadboard and then insert the 328.
- LED, standard 20mA
- Resistor 1kOhm
- Capacitor 100nF, 10V min, ceramic
- Potentiometer for tests
- Patch cables m-f, 5 pieces and some extra to connect RPI and breadboard
- wires for breadboard

Of course you need a RPi.

Setup procedure

Power off Rpi
Insert controller in breadboard.



VCC of the controller ifs from 3.3V from RPi (black, red).
SPI-connections MISO, MOSI, SCK and SS an CS0 des RPI (blue).
RESET of controller Pin 1 to GPIO24 (blue)
LED with series resistor is from controller PB1, Pin 15 to GND.
ADC-inputs are ADC0 und ADC1, pin 23, 24.

Input voltage range for the ADC is 0...3.3V.

Verkabelung überprüfen.
RPi einschalten.

Install software on Raspberry Pi

The procedures are for Raspbian distribution. You need root access to the system.

```
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get install python-dev pip
sudo pip install spidev intelhex
```

Activate SPI driver. Can be done with raspi-config (enable SPI) permanently.
Or with

```
sudo modprobe  spi_bcm2708
```

which needs to be repeated after each reboot.

Copy the software archive to /home/pi and unpack.

```
tar xzvf program_328.tar.gz
```

Verify hardware and programming software.

Read Atmega fuses

```
cd ~/program_328
sudo python src/program.py -rf
```

The output should read like this

```
PROGRAMMING_READ_CALIBRATION_BYTE   b0 10110000
PROGRAMMING_READ_EXTENDED_FUSE_BITS ff 11111111
   BODLEVEL0          1
   BODLEVEL1          1
   BODLEVEL2          1
PROGRAMMING_READ_FUSE_BITS         e2 11100010
   CKSEL0             0 ENABLED
   CKSEL1             1
   CKSEL2             0 ENABLED
   CKSEL3             0 ENABLED
   SUT0              0 ENABLED
   SUT1              1
   CKOUT              1
   CKDIV8             0
PROGRAMMING_READ_FUSE_HIGH_BITS     d9 11011001
   BOOTRST            1
   BOOTSZ0            0 ENABLED
   BOOTSZ1            0 ENABLED
   EESAVE             1
   WDTON              1
   SPIEN             0 ENABLED
   DWEN               1
   RSTDISBL           1
PROGRAMMING_READ_LOCK_BITS         ff 11111111
```

If there are error messages (e.g. device not in sync), then possibly there are wrong connections or the processor has fuses already programmed. Check wiring first.

Read current controller program

When output is correct, next step is to read out current flash program. Is empty on a new device, but either way useful to verify communication.

```
cd ~/program_328
sudo python src/program.py -r
```

Output is expected like this.

```
root@raspberrypi:/home/pi/program_328# python src/program.py -r
('read', 'out.hex')
programming_readCode
programming_enable
('PROGRAMMING_ENABLE', [172, 83, 0, 0])
(0, [255, 255, 83, 0])
programming_enable end
```

```
programming_disable
programming_readCode ende
ok
```

Flash 'blink' program

If successful, then load the first 'blink code' program into the controller. It will blink the LED.

```
cd ~/program_328
sudo python src/program.py -p 328/steckbrett_328_blink.hex
```

Takes a few seconds, and the LED should blink.

Program fuses to final settings

Everythink ok, then flash the fuses to have the 8MHz oscillator running.

```
cd ~/program_328
sudo python src/program.py -wf
```

Blinking will stop during the flash procedure, and restart noticeably faster, 5 times a second.
The controller has the internal clock divider disabled now, runs at 8MHz and wiring and software is ok.

Flash the firmware

Flash the firmware. It supports the various functions of the device.

```
cd ~/program_328
sudo python src/program.py -p 328/steckbrett_328.hex
```

Firmware Overview

The firmware features need to be enabled by configuration commands before using them.

These settings are not persistent, so after reset or reboot, these command need to be issued again.

For each Feature, there are SET_CONFIG and GET_CONFIG-commands.

The LED-commands are available without prior activation.

After reset, the LED will blink 8 times.

A note on the commands for the controller. When a response from controller is expected, there is the need to shift the according number of dummy bytes into the SPI. These bytes are indicated by trailing '0' after the commands.

SPI clock speed is set to 240.000 Hz for the 8MHz version of the firmware. For this speed, the spidev library leaves enough time between the bytes for the interrupt program to provide responses. At a higher speed, there will be communication problems. Lower speed is no problem.

Firmware, read version

GET_VERSION,0,0	0x80	Controller responds with version number.

Version of the firmware is 0x93, minor version depends on release.

```
cd ~/program_328
sudo python src/test_version.py
```

Display should be '0x93', minor version is e.g. 0x0C or higher.

Firmware: LED-Commands

SET_LED_0	0x84	drive Port PD7 (Pin13) low.
SET_LED_1	0x88	drive Port PD7 (Pin13) high.

Test program.
Led blinking is controlled from the raspberry. Rythm is long low, short dark.

```
cd ~/program_328
sudo python src/test_blink.py
```

Led on, off

```
cd ~/program_328
sudo python src/test_led_on.py
sudo python src/test_led_off.py
```

Firmware: Read ADC values

There are two ADC channels supported.

ADC-Conversions need to be enabled.

GET_ADC_0,0,0	0x81	get MSB,LSB ADC 0
GET_ADC_1,0,0	0x82	get MSB,LSB ADC 1
GET_ADC_CONFIG,0,0	0x91	Read configuration of ADC software. Default ist 0x03, 0x03 for AD0, AD1
SET_ADC_CONFIG, x, x	0x92	Write configuration bytes. First byte is AD0, second byte AD1 Bit 1 set: scale 3.3V Bit 1 clear: scale 1.1V Bit 0 set: activ Bit 0 clear: inactiv

The program reads some 100 values and prints to console.

```
cd ~/program_328
```

```
sudo python src/test_adc_0.py
```

If the input pin is unconnected, the readout will be 0 and eventually some 1,2,3 caused by noise..

ADC, Connection to Scratch

Main purpose of this device is to interface with scratch. Download and install scratch client software from heppg.de.

```
cd ~/scratchClient
sudo python src/scratchClient.py -config config/328_steckbrett.xml
```

The values from the two ad-channels are send as 'adc_0' und 'adc_1' to scratch geschickt, range 0..1023.
For this setup, the LED can be controlled by broadcast 'led_0_off', 'led_0_on'.
The channels being active, the scale and polling interval are configured in the adapter configuration xml.

Firmware: Event counter, frequency measurement

The event counter and frequency measurement are mutually exclusive. These functions all use the 16 bit Timer1 inside the controller.

On hardware side, these functions use either Pin T1 or ICP1, Pin 11, 14. Connecting these pins together allows greatest flexibility for the software.

The host is responsible to set the appropriate operation mode to achieve best accuracy.

First, check frequency range with Counter mode and two measurements in 1 sec distance.

For frequences > 20kHz use Counter mode.

For frequencies > 122 Hz and < 20kHz use timer mode.
Frequencies < 122 Hz can't be precisely measured in 10ms time slot.

Firmware: Event counter COUNTER

The internal timer1, 16 bits, is complemented by two bytes for the overflow. So 32 Bits are available.
Input Pin is T1.

Reading the results does not clear the registers.
Performance burden is low in this operation mode, as overflow is quite low frequent.

Firmware: Frequency measurement TIMER

The controller can measure frequencies.
Applicable for frequencies greater 200Hz and less than 20kHz

Input pin is ICP1.
Measurment is performed with capture function of timer1.

```
N = 4
repeat

      reset Timer 1
      wait for positive edge of signal, get t0-value
      Wait N periods, get tn-value.
      If overflow occurred, decrease N and discard values.
      Store tn-t0 and N and send on request to host.
      If value  > 0xF000, decrease N
      if value < 0xD000, increase N
```

Timer1 is using the clock frequency as time base for the measurement. At 8MHz the Duration of a measurement is 0xffff/8MHz = 8,2ms.
The host receives value (tn-t0), 16Bit and N.

Calculate the frequency:

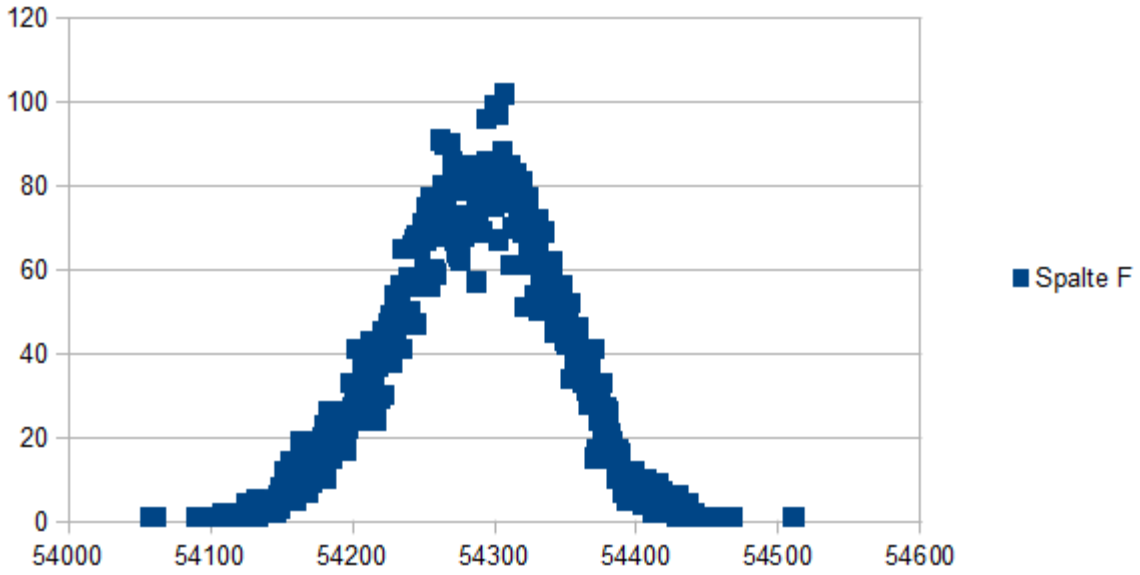
```
f = 8000000.0 *period / value
```

Validation of accuracy

I wanted to find out how accurate the measurements are and also have some test runs to find out whether there are bugs in firmware. Here the results.

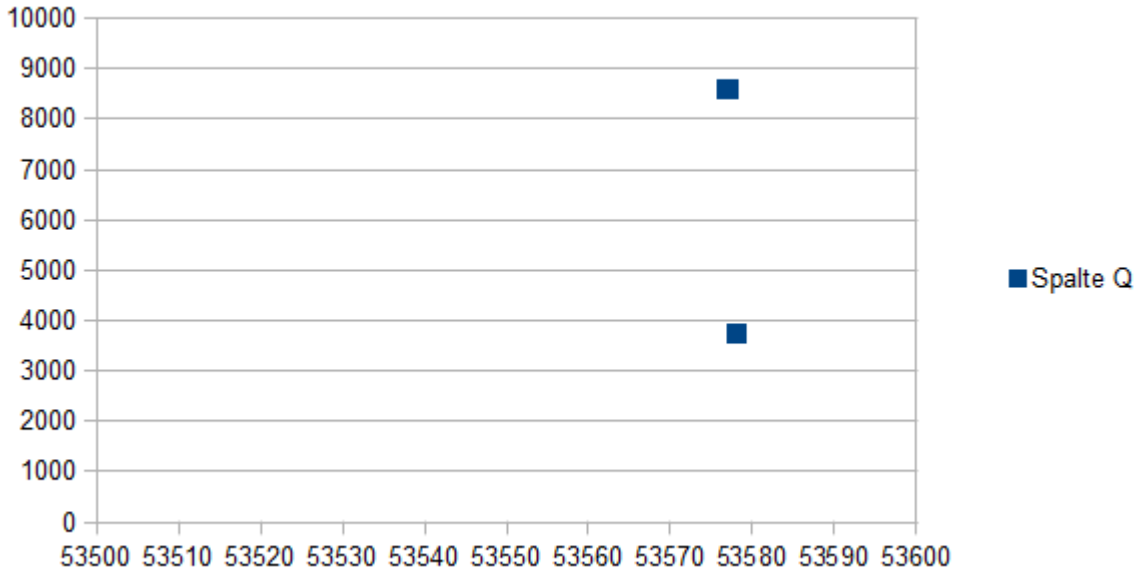
Precision of the measurement depends on the precision of the oscillator, here the internal RC-Oscillator.

I wanted to determine how accurate this measurement is and used a quartz stabilised input frequency with prox 10kHz and performed 10000 measurements. The following chart shows the values and accumulated number of occurence.



The x-axis is the the duration of 90 signal durations in processor clock time. The y-axis is the number of 'how often did this time value occur'. For a perfect internal oscillator, you woud expect only one sharp peak. From the distribution you see the precision is a few permille.

Same setup, but operating the controller with a quartz oscillator gives a perfect result. The result is only one bit difference, please note the different scale on y-axis..



The two points together Dsummarize more than 10.000 measurements.

SPI-Befehle for frequeny measurement are

`GET_TIMER, 0, 0, 0, 0`	0x87	Get results. Byte [1]: value high Byte [2]: value low Byte [3]: number of signal periods in measurement. Byte [4]: error, 0 = ok
`SET_COUNTER_CONFIG, 0, 0`	0x85	Byte[1].0 counter mode Byte[1].1 start counter Byte[1].2 reset counter Byte[1].3 frequency measurement mode Byte[1].4 enable noise canceller, recommended.
`GET_COUNTER_CONFIG, 0, 0`	0x86	Get current config.

The frequency measurement mode needs lot of performance. If signal frequency is too high, the controller will not be able to execute interrupt logic in time. I checked operation till 20kHz.

Firmware: Frequency measurement with a time base of 10ms, 20ms TIMEDCOUNTER

For frequencies higher than 20kHz it is appropriate to use event counting in a 10ms/ 20ms time slot.
Precisely, the 10ms slot is more like 9,98ms.
The t1-counter has 16 bits, theoretically you can measure up to 6.5MHz.

SPI-commands

`GET_TIMEDCOUNTER, 0, 0`	0x89	Read frequency measurement values. Byte [1]: value high Byte [2]: value low
`SET_COUNTER_CONFIG, 0, 0`	0x85	Byte[1].4 enable noise canceller, recommended. Byte[1].5 enable measurement. Byte[1].6 0: time slot 10ms 1: time slot 20ms
`GET_COUNTER_CONFIG, 0, 0`	0x86	Reads current configuration.

The results are the number of events in 10ms/20ms time slot (approximately). Accuracy depends on presicision of internal oscillator and is decreased by possible communication interrupts during the time slot.

Firmware: Infrared Control

The controller can receive infrared controller signals.

Use a TSOP34838 on PD6, Pin 12 to detect the infrared signals.

The controller does not decode the signals, but measures time between signal edges and provides these for the host.
The function needs to be enabled by configuration.

After receiving a signal, the controller uses Port PD4 to indicate availability of data (active low). The host will use GPIO#23 to detect this.
It is also possible to read status periodically (polling) from host.

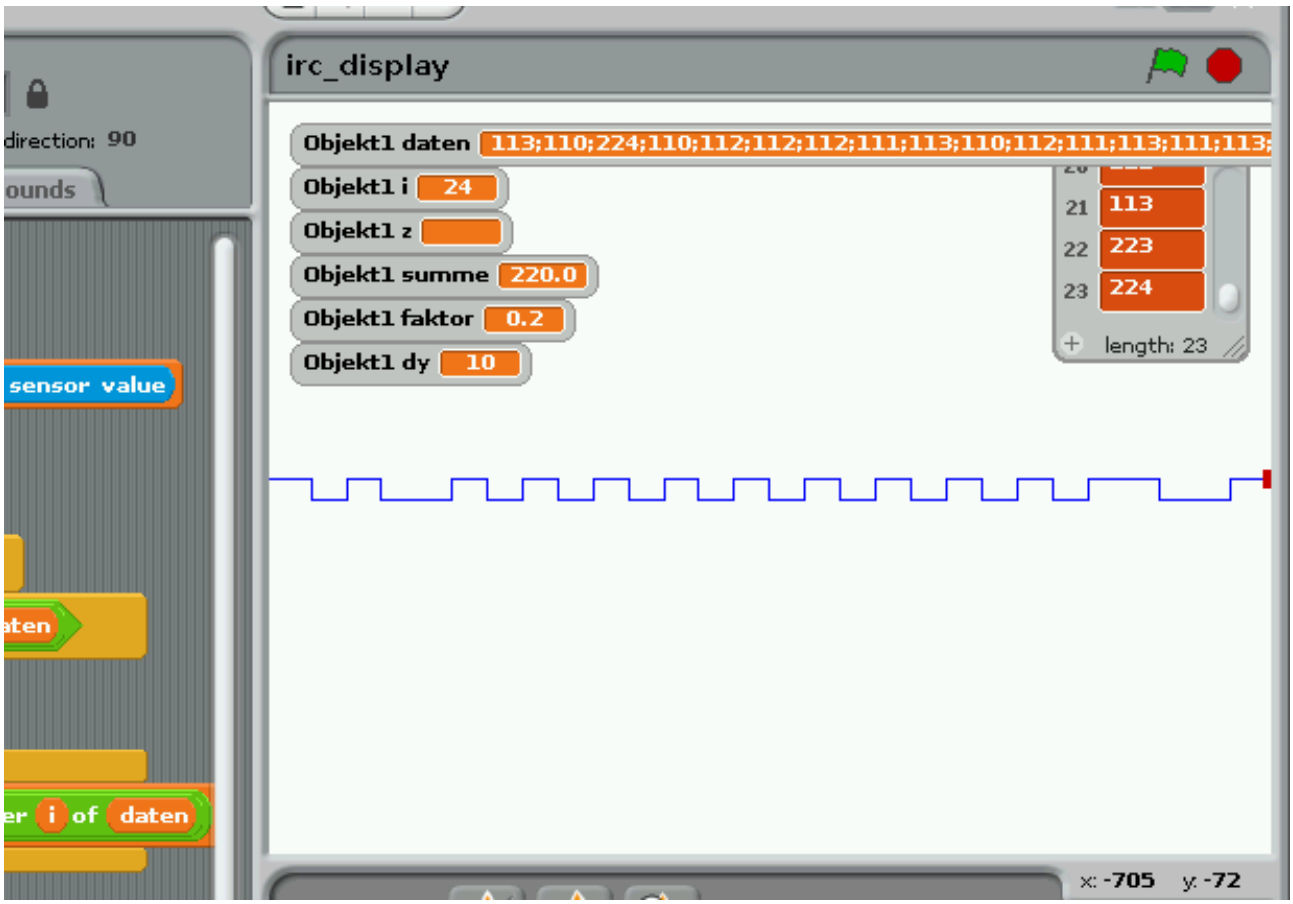
Data quisition is started by a negative edge (edge 0).
Data aquisition is for max 128 edges, each with 16 bit data. Resolution on 8MHz controller is 8us.
If there are no signal changes for more than 12.3ms, the sequence is assumed to be complete. Typical breaks between signls are 60 to 100 ms.

`SET_IRC_CONFIG, 0,`	0x8A	Write configuration. Byte [1].0: 1 = enable Byte [1].1: 1 start aquisition

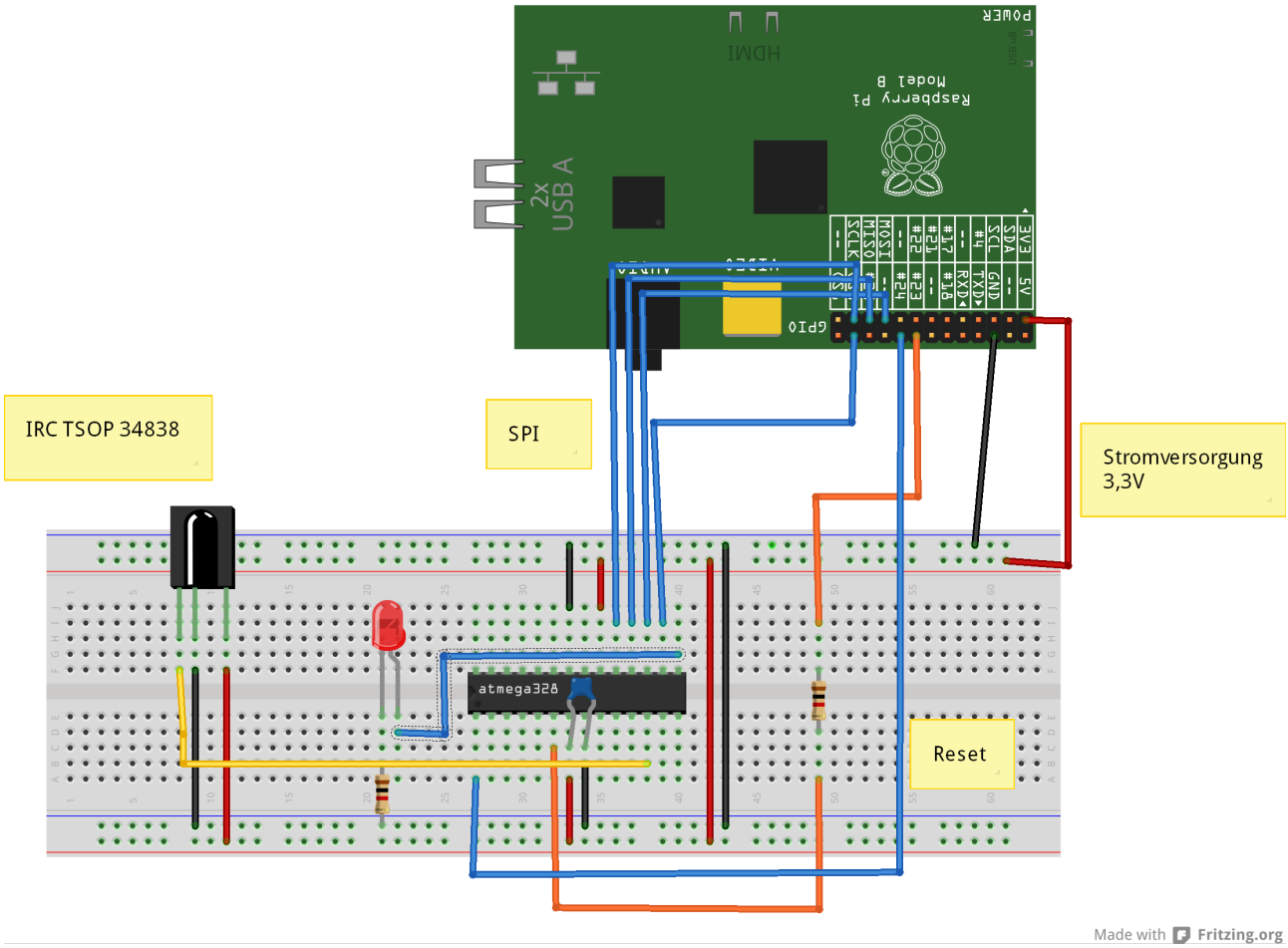
GET_IRC_DATA_0, 0, 0..(64 Bytes)	0x8B	Block 0 of results Byte[1] edge 1 MSB Byte[2] edge 1 LSB Byte[3] edge 2 MSB Byte[4] edge 2 LSB
GET_IRC_DATA_1, 0, 0 (64 Bytes)	0x8C	Block 1 of results
GET_IRC_DATA_2, 0, 0 (64 Bytes)	0x8D	Block 2 of results
GET_IRC_DATA_3, 0, 0 (64 Bytes)	0x8E	Block 3 of results
GET_IRC_STATUS, 0, 0, 0	0x8F	Read status. All bytes '0': no data available Byte 1: number of edges, MSB Byte 2: number of edges, LSB Byte 3: 2 == sequence complete 1 == Max number of edges

The program test_get_irc.py starts an aquisition, and tries to interpret the data as RC5. Older philips-remote control support this protocol. If the protocol can't be decoded, ther will be error messages.

There is a scratch-Program in the scratchClient-distribution which graphically displays the timing.



Configuration: /scratchClient/scratch/irc_atmel328/config_irc_atmel328.xml
Scratch-Program /scratchClient/scratch/irc_atmel328/irc_atmel328.sb



I used the TSOP34838 as it was available from my retailer. There are many similiar devices. Check frequency (38kHz), voltage (3.3V) and pinout (datasheet).
The 'orange' connection from Pin6 to GPIO#23 is connected by a 1kOhm Resistor in order to prevent damage in case of GPIO#23 is configured as an output.

Appendix: GPIO#4, RPi as Clockgenerator for Controller

An alternate function of GPIO#4-Pin is a clock output. The code to activate this is in then bin/-folder. It is based on code from guzunty.org.

Connect GPIO#4 with XTAL1-input , Pin 9 of controller.

Then start the program. Use a different terminal for this.

```
cd ~/program_328/bin
chmod +x gz_clock_1.92MHz
sudo ./gz_clock_1.92MHz
```

Output frequency is prox 1.92MHz.

Kepp this program running while you program the fuses.
Then disconnect the connection from GPIO#4 and terminate the program.