YoMoPi - Library

Stefan Jost

14. Februar 2018

Inhaltsverzeichnis

1 Imports

Following imports were needed for this library.
time: time was needed to get the timestamp on each value.
math: math was needed to calculate the squareroot for the reactive energy.
spidev: the spidev module was needed to use the SPI on the Raspberry Pi
sys:
RPi.GPIO: the RPi.GPIO was needed to controll the GPIO pins for the reset pin and further extentions

```
import time
import math
import spidev
import sys
import RPi.GPIO as GPIO
```

2 Classvariables

Following variable were needed to correcty access the register and to adjust the register values to usefull values.
read and **write**: the read and write were needed to mask the register address to not accidently write the register instead of reading it. Therefor to read a register the given address will be bitwise AND masked with the **read** variable and to write a register the address will be bitwise OR masked with the **write** variable.
spi, **active_lines** and **debug**: this variables were used to save the spi object, to save the number of actives phases that will be measured and to enable debugging via the console.
sampleintervall, **active_factor**, **apparent_factor**, **vrms_factor** and **irms_factor**: the sampleintervall defines the time between each sample when the start_sampling method is called. The minimum time between each sample is 1 second. Sampleintervall defines the number of seconds and can take each integer value greater then 0. The different factor variables will adjust the values read from the registers. This factors can be changes but should not be changed because they are calibrated values.

```
read = 0b00111111
write = 0b10000000
spi=0
active_lines = 1
debug = 1

sampleintervall = 1
active_factor = 1
apparent_factor = 1
vrms_factor = 1
irms_factor = 1
```

3 Methods

In this section every method from the library is listed and you will find a detailed description on the parameters and returns of each function. For more information you will also find the full source code of the function.

3.1 __init__

Description: This is the constructor and it creates a new YoMoPi object. It also creates a new SPI objects for each YoMoPi object.
Parameters: None.
Returns: Nothing.

```
def __init__(self):
        self.spi=spidev.SpiDev()
        return
```

3.2 init_yomopi

Description: Initializes the YoMoPi object. Sets the GPIO mode, disables GPIO warnings and defines pin 19 as output. Also opens a new SPI connection via the SPI device (0,0), sets the SPI speed to 62500 Hz and sets the SPI mode to 1. Finaly the function set_lines is called to set the MMODE, WATMODE and VAMODE.
Parameters: None.
Returns: Nothing.

```
def init_yomopi(self):
         GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(19,GPIO.OUT)
        self.spi.open(0,0)
        self.spi.max_speed_hz = 62500
        self.spi.mode = 0b01
         self.set_lines(self.active_lines)
        return
```

3.3 set_lines

Description: This function sets the number of active phases that will be measured.
Parameters: lines - 1 or 3
Returns: Nothing.

```
def set_lines(self, lines):
         if (lines != 1) and (lines != 3):
         print "Wrong␣number␣of␣lines"
                 return
         else:
                 self.active_lines = lines
                 if self.active_lines == 3:
                         self.write_8bit(0x0D, 0x3F)
```

```
                        self.write_8bit(0x0E, 0x3F)
                        self.set_mmode(0x70)
                elif self.active_lines == 1:
                        self.write_8bit(0x0D, 0x24)
                        self.write_8bit(0x0E, 0x24)
                        self.set_mmode(0x10)
                return
        return
```

3.4 enable_board

Description: Enables the board by pulling pin 19 into the HIGH state.
Parameters: None.
Returns: Nothing.

```
def enable_board(self):
        GPIO.output(19, GPIO.HIGH)
        return
```

3.5 disable_board

Description: Disables the board by pulling pin 19 into the LOW state.
Parameters: None.
Returns: Nothing.

```
def disable_board(self):
        GPIO.output(19, GPIO.LOW)
        return
```

3.6 write_8bit

Description: Writes 8 bit of data to the given address.
Parameters: register - 8 bit address of the register (see ADE7754 register table)
value - 8 bit of value that will be written into the register
Returns: Nothing.

```
def write_8bit(self, register, value):
        self.enable_board()
        register = register | self.write
        self.spi.xfer2([register, value])
        return
```

3.7 read_8bit

Description: Reads 8 bit of data from the given address.
Parameters: register - 8 bit address of the register (see ADE7754 register table)
Returns: the 8 bit of data in the register as decimal

```
def read_8bit(self, register):
        self.enable_board()
        register = register \& self.read
        result = self.spi.xfer2([register, 0x00])[1:]
        return result[0]
```

3.8 read_16bit

Description: Reads 16 bit of data from the given address.
Parameters: register - 8 bit address of the register (see ADE7754 register table)
Returns: the 16 bit of data in the register as decimal

```
def read_16bit(self, register):
        self.enable_board()
        register = register \& self.read
        result = self.spi.xfer2([register, 0x00, 0x00])[1:]
        dec_result = (result[0]<<8)+result[1]
        return dec_result
```

3.9 read_24bit

Description: Reads 24 bit of data from the given address.
Parameters: register - 8 bit address of the register (see ADE7754 register table)
Returns: the 24 bit of data in the register as decimal

```
def read_24bit(self, register):
        self.enable_board()
        register = register \& self.read
        result = self.spi.xfer2([register, 0x00, 0x00, 0x00])[1:]
        dec_result = (result[0]<<16)+(result[1]<<8)+(result[0])
        return dec_result
```

3.10 get_temp

Description: Reads 8 bit of data from the temperature register (0x08).
Parameters: None.
Returns: A list of two elements [timestamp, temperature in °C]

```
def get_temp(self):
        reg = self.read_8bit(0x08)
        temp = [time.time(),(reg-129)/4]
        return temp
```

3.11 get_aenergy

Description: Reads 24 bit of data from the active ernergy register (0x02) and resets the register to 0.
Parameters: None.
Returns: A list of two elements [timestamp, value of the register]

```
def get_aenergy(self):
       aenergy = [time.time(), self.read_24bit(0x02)]
       return aenergy
```

3.12 get_appenergy

Description: Reads 24 bit of data from the apparent ernergy register (0x05) and resets the register to 0.
Parameters: None.
Returns: A list of two elements [timestamp, value of the register]

```
def get_appenergy(self):
        appenergy = [time.time(), self.read_24bit(0x05)]
        return appenergy
```

3.13 get_period

Description: Reads 16 bit of data from the period register (0x07).
Parameters: None.
Returns: A list of two elements [timestamp, value of the register]

```
def get_period(self):
       period = [time.time(), self.read_16bit(0x07)]
       return period
```

3.14 set_opmode

Description: Sets the OPMODE. For more information to the OPMODE see section OPMODE.
Parameters: value - 8 bit of data representing the OPMODE
Returns: Nothing.

```
def set_opmode(self, value):
        self.write_8bit(0x0A, value)
        return
```

3.15 set_mmode

Description: Sets the MMODE. For more information to the MMODE see section MMODE.
Parameters: value - 8 bit of data representing the MMODE
Returns: Nothing.

```
def set_mmode(self, value):
        self.write_8bit(0x0B, value)
        return
```

3.16 get_sample

Description: Takes one sample and calculates the active energy, apparent energy, reactive energy, VRMS and IRMS. The calculated values are the real values. (after adjusting with their factores)
Parameters: None.
Returns: A list of 7 elements [timestamp, active energy, apparent energy, reactive energy, period, VRMS, IRMS]

```
def get_sample(self):
        aenergy = self.get_aenergy()[1] *self.active_factor
        appenergy = self.get_appenergy()[1] *self.apparent_factor
        renergy = math.sqrt(appenergy*appenergy - aenergy*aenergy)
        if self.debug:
                print "Active␣energy:␣%f␣W,␣Apparent␣energy:␣%f␣VA,␣Reactive␣Energy:␣%f␣var"
                % (aenergy, appenergy, renergy)
                print "VRMS:␣%f␣IRMS:␣%f"
                %(self.get_vrms()[1]*self.vrms_factor,self.get_irms()[1]*self.irms_factor)
        sample = []
        sample.append(time.time())
        sample.append(aenergy)
        sample.append(appenergy)
        sample.append(renergy)
        sample.append(self.get_period()[1])
        sample.append(self.get_vrms()[1]*self.vrms_factor)
        sample.append(self.get_irms()[1]*self.irms_factor)
        return sample
```

3.17 get_vrms

Description: Reads the VRMS register depending on if the active lines is 1 or 3.
Parameters: None.
Returns: A list of 2 elements [timestamp, Phase A VRMS] or 4 elements [timestamp, Phase A VRMS, Phase B VRMS, Phase C VRMS]

```
def get_vrms(self):
        if self.active_lines == 1:
                avrms = [time.time(), self.read_24bit(0x2C)]
                return avrms
        elif self.active_lines == 3:
              vrms = []
              vrms.append(time.time())
                vrms.append(self.read_24bit(0x2C))
                vrms.append(self.read_24bit(0x2D))
                vrms.append(self.read_24bit(0x2E))
                return vrms
        return 0
```

3.18 get_irms

Description: Reads the IRMS register depending on if the active lines is 1 or 3.

Parameters: None.
Returns: A list of 2 elements [timestamp, Phase A IRMS] or 4 elements [timestamp, Phase A IRMS, Phase B IRMS, Phase C IRMS]

```
def get_irms(self):
        if self.active_lines == 1:
                airms = [time.time(), self.read_24bit(0x29)]
                return airms
        elif self.active_lines == 3:
              irms = []
              irms.append(time.time())
                irms.append(self.read_24bit(0x29))
                irms.append(self.read_24bit(0x2A))
                irms.append(self.read_24bit(0x2B))
                return vrms
        return 0
```

3.19 start_sampling

Description: Starts a sampling programm that takes a number of samples depending on the parameters.
Parameters: nr_samples - this are the number of samples that will be taken, integer greater then 0
samplerate - this is the time between each sample, integer greater then 0
Returns: A list of samples (each sample is a list of 7 elements)

```
def start_sampling(self, nr_samples, samplerate):
        if (samplerate<1) or (nr_samples<1):
                return 0
        self.sampleintervall = samplerate
        samples = []
        for i in range(0, nr_samples):

                for j in range(0, samplerate):
                        time.sleep(1)

                samples.append(self.get_sample())
        return samples
```

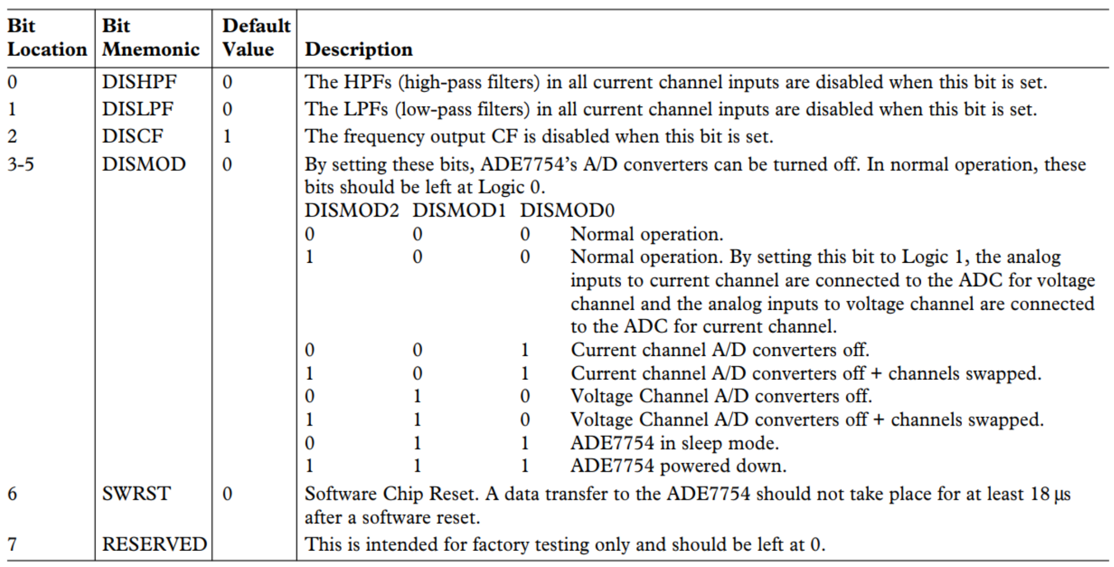
3.20 close

Description: Closes the SPI connection.
Parameters: None.
Returns: Nothing.

```
def close(self):
        self.spi.close()
        return
```

4 OPMODE

The general configuration of the ADE7754 is defined by writing to the OPMODE register (0x0A).

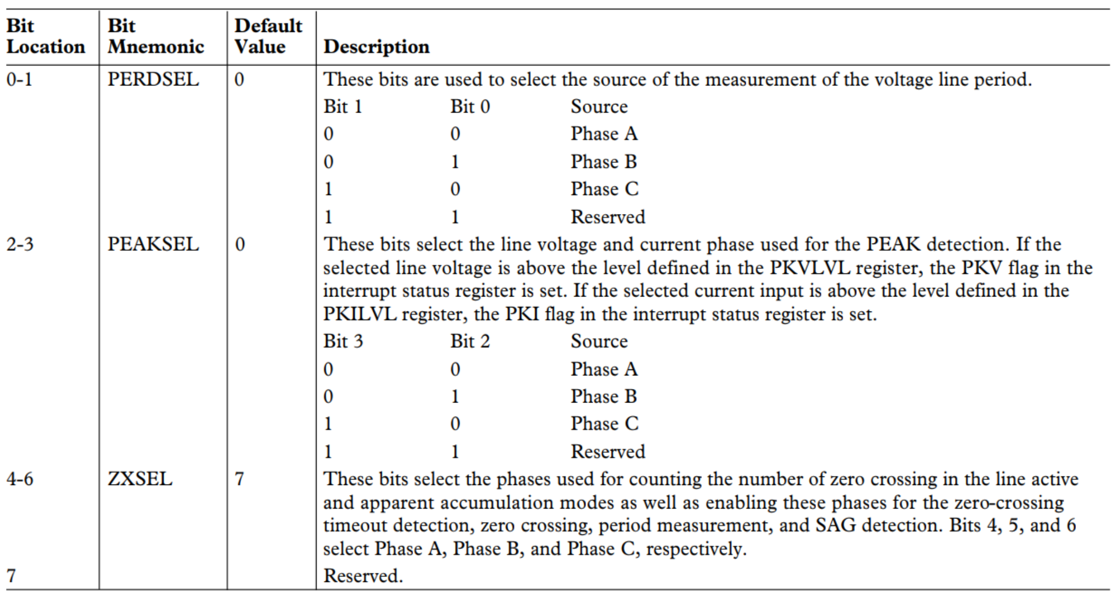


Bit Location	Bit Mnemonic	Default Value	Description
0	DISHPF	0	The HPFs (high-pass filters) in all current channel inputs are disabled when this bit is set.
1	DISLPF	0	The LPFs (low-pass filters) in all current channel inputs are disabled when this bit is set.
2	DISCF	1	The frequency output CF is disabled when this bit is set.
3-5	DISMOD	0	By setting these bits, ADE7754's A/D converters can be turned off. In normal operation, these bits should be left at Logic 0. DISMOD2 DISMOD1 DISMOD0 0 0 0 Normal operation. 1 0 0 Normal operation. By setting this bit to Logic 1, the analog inputs to current channel are connected to the ADC for voltage channel and the analog inputs to voltage channel are connected to the ADC for current channel. 0 0 1 Current channel A/D converters off. 1 0 1 Current channel A/D converters off + channels swapped. 0 1 0 Voltage Channel A/D converters off. 1 1 0 Voltage Channel A/D converters off + channels swapped. 0 1 1 ADE7754 in sleep mode. 1 1 1 ADE7754 powered down.
6	SWRST	0	Software Chip Reset. A data transfer to the ADE7754 should not take place for at least 18 µs after a software reset.
7	RESERVED		This is intended for factory testing only and should be left at 0.

Abbildung 1: OPMODE

5 MMODE

The configuration of the period and peak measurements made by the ADE7754 are defined by writing to the MMODE register (0x0B).



Bit Location	Bit Mnemonic	Default Value	Description
0-1	PERDSEL	0	These bits are used to select the source of the measurement of the voltage line period. Bit 1 / Bit 0 / Source 0 / 0 / Phase A 0 / 1 / Phase B 1 / 0 / Phase C 1 / 1 / Reserved
2-3	PEAKSEL	0	These bits select the line voltage and current phase used for the PEAK detection. If the selected line voltage is above the level defined in the PKVLVL register, the PKV flag in the interrupt status register is set. If the selected current input is above the level defined in the PKILVL register, the PKI flag in the interrupt status register is set. Bit 3 / Bit 2 / Source 0 / 0 / Phase A 0 / 1 / Phase B 1 / 0 / Phase C 1 / 1 / Reserved
4-6	ZXSEL	7	These bits select the phases used for counting the number of zero crossing in the line active and apparent accumulation modes as well as enabling these phases for the zero-crossing timeout detection, zero crossing, period measurement, and SAG detection. Bits 4, 5, and 6 select Phase A, Phase B, and Phase C, respectively.
7			Reserved.

Abbildung 2: MMODE

6 Examples

To use the YoMoPi library you first need to import the library.

```
import yomopi
```

When the library is correcty imported you can create a new YoMoPi object and initialize is with the init_yomopi function.

```
yomo = yomopi.YoMoPi()
yomo.init_yomopi()
```

To test the implementation we start a short sampling. Therefor we call the start_-sampling function with 5 samples and a delay of 1 second between each sample.

```
yomo.start_sampling(5, 1)
```

If you want to switch the measurement to one phase change the line like this.

```
yomo.set_lines(1)
```

Abbildungsverzeichnis