
ADS1256

Arduino Library

DOCUMENTATION AND MANUAL

CURIOUS SCIENTIST

[CURIUSSCIENTIST.TECH](https://curiousscientist.tech)

LAST UPDATED MARCH 28, 2025

Contents

1	Introduction	2
2	ADS1256 Analog-to-digital converter	3
3	Registers	6
3.1	Reading a register	6
3.2	Writing a register	8
3.3	Status register - Address: 0	9
3.4	MUX register - Address: 1	11
3.5	ADCON register - Address: 2	13
3.6	DRATE register - Address: 3	17
3.7	I/O register - Address: 4	18
3.8	Calibration registers - Address 5-10	22
4	Coding	24
4.1	Constructor	24
4.2	Handling the DRDY signal	26
4.3	Initialization	27
4.4	Converting the ADC count into a proper format	29
4.5	Converting the acquired value into voltage	29
4.6	Reading a single conversion	30
4.7	Reading a single input continuously	31
4.8	Stopping continuous conversions	32
4.9	Multiplexing through the 8 single-ended inputs	33
4.10	Multiplexing through the 4 differential inputs	36
5	Testing the code	39
5.1	Precision	39
5.1.1	With input buffer	39
5.1.2	Without input buffer	39
5.2	Sampling rate measurement	40
6	List of tested microcontrollers	42
7	Versions and changelog	43
7.1	First release - 2022-07-14	43
7.2	Changed PayPal link - 2023-01-20	43
7.3	Updates - 2023-11-10	43
7.4	Updates - 2024-02-08	43
7.5	Updates - 2024-09-12	44
7.6	Updates - 2025-03-28	45

1 Introduction

I created this documentation to summarize and explain the functionalities of my custom-made library for the ADS1256 24-bit, 8 channel analog-to-digital converter (ADC). This ADC can provide 30000 samples per second (SPS) data rate on a single channel and it can run at 4374 Hz when the inputs are cycled via the input multiplexer. The ADC communicates via SPI, so it is easy to connect it to any of today's popular microcontroller units (MCUs). Since the ADS1256 uses the SPI, the user has to consult the datasheet of the MCU for the SPI pins (MISO, MOSI and SCK pins). Other, application-specific pins (DRDY, CS, SYNC/PDWN and RESET) are further discussed in this document.

In this document I am going to show how each functions are implemented for Arduino-compatible microcontrollers. I personally prefer to use MCUs with a native USB support. For example ATmega32U4, Teensy 4.0, STM32F103C8T6 ("*blue pill*"), RP2040...etc.

I also designed a printed circuit board (PCB) that utilizes an RP2040 microcontroller and I think it is just fine to drive this ADC ¹. You can buy the PCB from PCBWay using my [affiliate link](#).

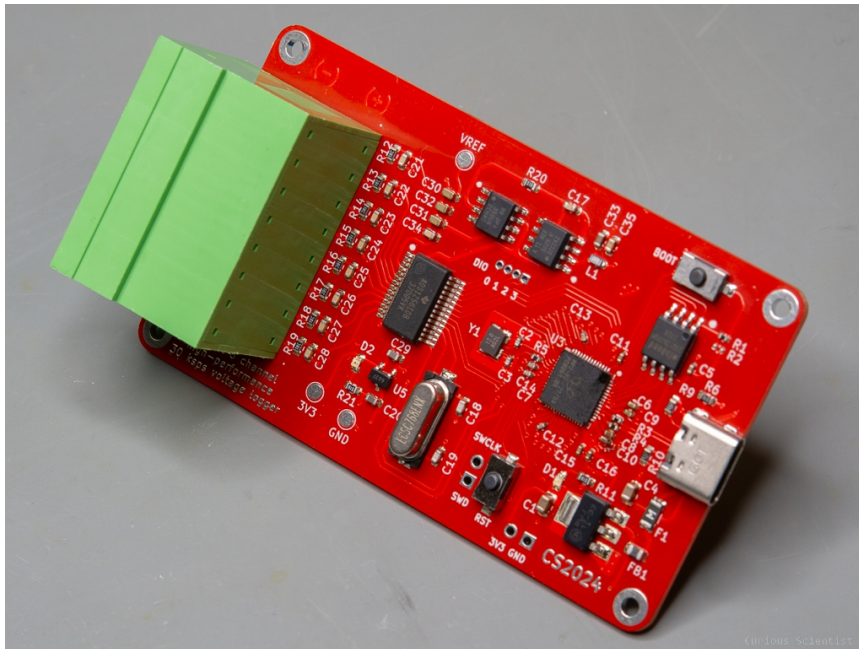


Figure 1: Custom-made ADS1256 data logger board driven by an RP2040 microcontroller.

If you think that this document helped your work, please consider supporting me by becoming a [YouTube Channel Member](#) or donate a voluntary amount via [PayPal](#).

If you use the document or the code for your publications, don't forget to cite this document:

Curious, Scientist. ADS1256 Arduino Library Documentation And Manual. 2022.
<https://curiousscientist.tech/ads1256-custom-library>

Disclaimer:

I have written this document and the library and made the circuit with honest and good intentions. Despite my best efforts, mistakes and errors can occur. In any case, I am not responsible for any damage, harm caused by the use of the published resources.

¹I am not an electrical engineer, I am just an enthusiastic scientist, so any recommendations to improve the design is welcome!

2 ADS1256 Analog-to-digital converter

This chip is made by Texas Instruments. I am not going to detail all of its functions here, so if you want to dive deeper than this document, please refer to the official datasheet of the chip[1].

This ADC has 8 input channels. They can be used in single-ended mode which means that the positive terminal is one of these 8 input channels and the negative terminal is the common ground connection. Another mode to utilize the input channels is to use them in differential mode. In this case both the positive and negative terminals are one of the 8 input channels, thus the total number of input channels is 4 in this case. The inputs have a buffer circuit which significantly increases the input impedance of the device but limits the input voltage. In typical applications where +5 V is the supply voltage, using the buffer would lead to 0 – 3 V measurement range. Of course, with proper precision voltage dividers[2], one can extend the measurement range while still maintaining good resolution. There is also a way to go the other way around: one can utilize the built in programmable gain amplifier (PGA) which provides more resolution when measuring smaller input signals (see Table 1). When using the PGA it is always recommended to use the one closest to the measured voltage. For example, if the maximum measured voltage range is ± 1 V, $\text{PGA} = 4$ is recommended.

PGA	Full-scale input voltage (V_{in}) ($V_{\text{ref}} = 2.5$ V)
1	± 5 V
2	± 2.5 V
4	± 1.25 V
8	± 625 mV
16	± 312.5 mV
32	± 156.25 mV
64	± 78.125 mV

Table 1: ADS1256 PGA settings and the corresponding input voltage ranges.

As it was mentioned in the first section, this chip is capable of measuring one channel at 30 kSPS data rate. Further, slower data rates (and higher resolution) can be achieved by applying the programmable filter of the circuit. The data rate is provided by the following equation:

$$\text{DRATE} = \left(\frac{f_{\text{clk in}}}{256} \right) \cdot \left(\frac{1}{\text{AVG}} \right), \quad (1)$$

where DRATE is the data rate in SPS, $f_{\text{clk in}}$ is the crystal frequency (typically, $f_{\text{clk in}} = 7.68$ MHz), and AVG is the number of averages taken of the reading.

The available sampling rates, filter values and register values (binary and decimal) are shown in Table 2. I also added the decimal equivalent of the binary values to the table because the original datasheet of the ADS1256 only published the register values in binary numbers, however it is easier for us to send a decimal number to the serial terminal to change the register setting.

Data rate (SPS)	Number of averages	Register value (binary)	Register value (decimal)
30000	1 (no averaging)	1111000	240
15000	2	1110000	224
7500	4	11010000	204
3750	8	11000000	192
2000	15	10110000	176
1000	30	10100001	161
500	60	10010010	146
100	300	10000010	130
60	500	01110010	114
50	600	01100011	99
30	1000	01010011	83
25	1200	01000011	67
15	2000	00110011	51
10	3000	00100011	35
5	6000	00010011	19
2.5	12000	00000011	3

Table 2: ADS1256 DRATE settings and corresponding register values.

The above table is only valid for single-channel data acquisition. If we want to utilize all the 8 (or 4) channels, the input multiplexer has to be used to cycle through the inputs. This involves several steps which will make sure that while the data is being retrieved, the next input is already prepared for capturing. This process has a downside: the data rate significantly drops. Up to 100 SPS we can get the same data rates as we set using the DRATE register, but above these values, due to the cycling, the effective DRATE becomes lower, and the maximum DRATE at 30000 SPS becomes 4374 Hz. The list of DRATE vs. throughput values are shown in Table 3 below.

Data rate (SPS)	Cycling throughput (Hz)
30000	4374
15000	3817
7500	3043
3750	2165
2000	1438
1000	837
500	456
100	98
60	59
50	50
30	30
25	25
15	15
10	10
5	5
2.5	2.5

Table 3: ADS1256 Multiplexer throughput values for different DRATE settings.

Once we have the PGA and DRATE values, we can proceed to acquire data. To be able to interpret the data, we need to know a little about the data format. The data is a 24-bit number which we capture in three, 8-bit packets (essentially, 3 bytes). The full process is shown in Section 4.6. The LSB has a weight of

$$\text{LSB} = \frac{2 \cdot V_{\text{ref}}}{\text{PGA} \cdot 2^{23} - 1}, \quad (2)$$

where V_{ref} is the value of the reference voltage and PGA is the selected PGA value (1, 2, 4..., 64).

A positive full-scale input produces an output code of 7FFFFFFh (decimal: 8388607) and the negative full scale input produces an output code of 8000000h (decimal: 8388608).

$$8388607 + 8388608 = 2^{24} - 1$$

Side note: For precise measurements, it is crucial to precisely know the value of the reference voltage (V_{ref}). If V_{ref} is not precisely 2.500000 V, don't use 2.5 V as the value of the reference voltage in the formulas. Try to measure V_{ref} with a good, preferably calibrated multimeter. I added a tab on my PCB for probing the output of the voltage reference chip. Commercial boards typically don't have this feature.

3 Registers

Reading a register is one of the fundamental tasks that we have to do when using the ADS1256 with a microcontroller. There are 11 registers in total and all of them can be accessed by communicating with the ADC via the SPI. Reading a register allows us to check if we have the correct settings on the ADS1256.

3.1 Reading a register

Reading a register is done by two command bytes (2×8 bits):

- 1st byte: 0001 *rrrr*, where *rrrr* is the address of the register to read
- 2nd byte: 0000 *nnnn*, where *nnnn* is the number of bytes to read minus one

With the above instructions we can easily construct the first byte by applying a logical OR operation between the 0001 0000 byte² and the address of the register to read (*rrrr*).

For example: 0001 0000 | 0000 0011 = 0001 0011 command will result in reading the 4th register which is the DRATE register. Also, the 2nd byte might be a little bit tricky to understand. For example, if we want to read only one register's value at a time, the 2nd byte has to be 0: 0000 0000. If we want to read two registers, it has to be 1: 0000 0001 and so on. This is the "minus one" part mentioned above.

The reading of a register is implemented in the following way:

```
1 long ADS1256::readRegister(uint8_t registerAddress)
2 {
3     waitForLowDRDY();
4     SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
5     digitalWrite(_CS_pin, LOW);
6     SPI.transfer(0x10 | registerAddress);
7     SPI.transfer(0);
8     delayMicroseconds(5);
9     _registerValuetoRead = SPI.transfer(0xFF);
10    digitalWrite(_CS_pin, HIGH);
11    SPI.endTransaction();
12    delay(100);
13    return _registerValuetoRead;
14 }
```

The function works in the following way after the *waitForLowDRDY()* function finished:

- 1.) The SPI communication is initiated
- 2.) The chip select pin is pulled to low to indicate the beginning of the SPI transaction
- 3.) The 1st command byte (0x10) combined with the register address is sent to the ADC
- 4.) The 2nd command byte (0x00) is sent to the ADC separately
- 5.) We wait t_6 , then send an empty byte to the ADC to shift out the value of the register
- 6.) We pull the chip select pin up and finish the SPI transaction
- 7.) After waiting 100 ms, the function returns the obtained register value

²I added a space in the middle of the number to make it easier to read. When you use the numbers in binary format, avoid using spaces!

In my library, I created a shorthand definiton for all the 11 registers in the ads1256.h header file, so they are much easier to call.

```
1  #define STATUS_REG 0x00
2  #define MUX_REG 0x01
3  #define ADCON_REG 0x02
4  #define DRATE_REG 0x03
5  #define IO_REG 0x04
6  #define OFC0_REG 0x05
7  #define OFC1_REG 0x06
8  #define OFC2_REG 0x07
9  #define FSC0_REG 0x08
10 #define FSC1_REG 0x09
11 #define FSC2_REG 0x0A
```

For example, if we want to read the DRATE register to find out the sampling rate, we can send the following command to the MCU and the code will return the value of the register:

```
1  readRegister(DRATE_REG);
```

The return value will be the decimal equivalent value of the register. For example, the return value 114 corresponds to 60 SPS sampling speed.

3.2 Writing a register

The code snippet below is the function that allows us to write a specific value to a specific register. The function does not have any return value, it only performs a task. The function expects two 8-bit integers (2 bytes) as the parameter: the address of the register to be written and the value to be written to the register.

- 1st byte: 0101 *www*, where *www* is the address of the register to be written
- 2nd byte: 0000 *nnnn*, where *nnnn* is the number of bytes to read minus one
- 3rd byte: 0000 0000, this is the desired value of the register

```
1 void ADS1256::writeRegister(uint8_t registerAddress, uint8_t registerValueToWrite)
2 {
3     waitForLowDRDY();
4     SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
5     digitalWrite(_CS_pin, LOW);
6     delayMicroseconds(5);
7     SPI.transfer(0x50 | registerAddress);
8     SPI.transfer(0);
9     SPI.transfer(registerValueToWrite);
10    digitalWrite(_CS_pin, HIGH);
11    SPI.endTransaction();
12    delay(100);
13 }
```

The function works in the following way after the *waitForLowDRDY()* function finished:

- 1.) The SPI communication is initiated
- 2.) The chip select pin is pulled to low to indicate the beginning of the SPI transaction
- 3.) The 1st command byte (0x50) combined with the register address is sent to the ADC
- 4.) The 2nd command byte (0x00) is sent to the ADC separately.
- 5.) Then we send the register value to the ADC
- 6.) We pull the chip select pin up and finish the SPI transaction
- 7.) We wait 100 ms and exit the function

A quick way of using the implemented function can be the following:

```
1 writeRegister(DRATE_REG, DRATE_50SPS);
```

This line will set the sampling rate to 50 SPS by writing the equivalent 8-bit value to the DRATE register.

You can also directly pass integer numbers to this function:

```
1 writeRegister(1, 35);
```

The above command will write the value 35 (0010 0011) to the 1st register. The 1st register is the MUX register and 35 (0010 0011) means we want to select the A2 and A3 pins as the (differential) input pins.

3.3 Status register - Address: 0

This is the first register in series with the address of zero. Its value map is really simple because 5 out of the 8 bits are read-only.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ID	ID	ID	ID	ORDER	ACAL	BUFEN	DRDY

Table 4: Status register and its contents.

Bit 7-4 are read-only bits, so they are omitted here. Bit 3 decides the order of the output bits: 0 - MSB (default), 1 - LSB. Bit 2 is the auto-calibration bit: 0 - disabled (default), 1 - enabled. Bit 1 is the analog input buffer: 0 - disabled (default), 1 - enabled. The last bit, bit 0 replicates the state of the DRDY pin, this is also a read-only pin.

I typically use the following settings:

- ORDER: 0 - MSB
- ACAL: 1 - Enabled
- BUFEN: 1 - Enabled

By combining these settings, we get the following byte: 0000 0110 (decimal = 6).

I made six separate routines to individually get and set the ORDER bit, the ACAL bit, and the BUFEN bit. They can be called in the following way:

```
1  setByteOrder(0);
2  getByteOrder();
3  setAutoCal(0);
4  getAutoCal();
5  setBuffer(0);
6  getBuffer();
```

Each of the set functions accept either 0 or 1 in the argument. The effect of these numbers are already explained above. When the get functions are used, they will print a text on the serial terminal. The get functions don't accept any arguments.

One of the above three set functions is shown below. If the *setByteOrder()* function receives a 0 or 1, it will overwrite the third bit of the `_STATUS` variable which stores the value of the STATUS register, then it will also send this value to the STATUS register. It is important that we first read the value from the status register, so we work with its most recent value.

```
1 void ADS1256::setByteOrder(uint8_t byteOrder)
2 {
3     _STATUS = readRegister(STATUS_REG);
4
5     if(byteOrder == 0)
6     {
7         bitWrite(_STATUS, 3, 0);
8     }
9     else if(byteOrder == 1)
10    {
11        bitWrite(_STATUS, 3, 1);
12    }
13    else{}
14    writeRegister(STATUS_REG, _STATUS);
15    delay(100);
16 }
```

Also, let's have an example of one of the three get functions. The function below retrieves the value of the whole status register and then it returns the third bit that determines whether the MSB or the LSB byte order is enabled.

```
1 void ADS1256::getByteOrder()
2 {
3     uint8_t statusValue = readRegister(STATUS_REG);
4
5     return bitRead(statusValue, 3);
6 }
```

3.4 MUX register - Address: 1

This is the input multiplexer register that allows us to select arbitrary terminals as the inputs of the ADC.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pos 3	Pos 2	Pos 1	Pos 0	Neg 3	Neg 2	Neg 1	Neg 0

Table 5: MUX register and its values.

Bit 7-4 select the positive input channel, and bit 3-0 select the negative input channel. By carefully combining the input channels we can create either 8 single-ended inputs (AIN0-AIN7 + AINCOM) or 4 differential inputs (AIN0-7 + AIN0-7). The inputs are not read simultaneously, but one after another. For example, first we define an input pin-pair (A0+A1), then we read a conversion, then we create another input pin-pair (A2+A3), and read another conversion, now from this new input...and so on. With this method we scan through the input channels and read them in succession. This is essentially the multiplexing.

Bit 7-4 or Bit 3-0	Input
0000	AIN0
0001	AIN1
0010	AIN2
0011	AIN3
0100	AIN4
0101	AIN5
0110	AIN6
0111	AIN7
1111	AINCOM (GND)

Table 6: MUX register values and corresponding input pins.

In Table 6 the possible values of the bits are shown. The bit 7-4 determine the positive input pin (e.g. AIN0). The 3-0 bits determine the negative pin (e.g. AINCOM (GND)). It should also be obvious that the positive and negative inputs cannot be the same channel!

A full command byte, for example *0010 0011* looks like this:

$$\begin{array}{ccc} & \text{Negative input is AIN3} & \\ & \underbrace{\hspace{1.5cm}} & \\ & 0011 & \\ \underbrace{\hspace{1.5cm}} & & \\ 0010 & & \end{array} \quad (3)$$

Positive input is AIN2

An example of selecting a single-ended input pair where the positive channel is AIN2 and the negative channel is AINCOM (GND):

```
1 writeRegister(MUX_REG, SING_2);
```

An example of selecting a differential input pair where the positive channel is AIN2 and the negative channel is AIN3:

```
1 writeRegister(MUX_REG, DIFF_2_3);
```

I have written predefined differential input channel-pairs in a sensible way by pairing up adjacent input pins (A0+A1, A2+A3...etc.). However, nothing hinders you from, for example creating a pair where the positive input channel is A2 (0010) and the negative input channel is A6 (0110). In this case, you have to manually create the command for the MUX register:

```
1 writeRegister(MUX_REG, B00100110);
```

In my library I predefined eight single-ended, and four differential input channels. The arrangement of the single-ended channels is straightforward because they are always a combination of one of the AIN_x ($x = 0 - 7$) pin (positive) and the AINCOM pin (negative, GND).

```
1 #define SING_0 0b00001111 //A0 + GND
2 #define SING_1 0b00011111 //A1 + GND
3 #define SING_2 0b00101111 //A2 + GND
4 #define SING_3 0b00111111 //A3 + GND
5 #define SING_4 0b01001111 //A4 + GND
6 #define SING_5 0b01011111 //A5 + GND
7 #define SING_6 0b01101111 //A6 + GND
8 #define SING_7 0b01111111 //A7 + GND
```

The differential channels are somewhat arbitrary but they still make sense. I decided to pair the adjacent inputs as differential input pairs which means that $\text{AIN0}+\text{AIN1}$ becomes an input pair, $\text{AIN2}+\text{AIN3}$ becomes an input pair, and so on. They are defined in the `ads1256.h` header file and they can be called as it is shown below.

```
1 #define DIFF_0_1 0b00000001 //A0 + A1
2 #define DIFF_2_3 0b00100011 //A2 + A3
3 #define DIFF_4_5 0b01000101 //A4 + A5
4 #define DIFF_6_7 0b01100111 //A6 + A7
```

I further simplified the procedure of modifying the MUX register to select the input pins by writing my own routine. With this simplified routine one only needs to call the `setMUX()` function and provide one of the above defined pin definitions (e.g. `SING_5`) in the argument, like it is shown below:

```
1 setMUX(SING_5);
```

The above command will select the single-ended input as $\text{AIN5} + \text{AINCOM}$ (GND).

The `setMUX()` function also accepts integers. If you know the decimal (or binary) value for the register to select a certain input, you can also just pass a number to the function. For example, the example below selects $\text{A4}+\text{A5}$ as input:

```
1 setMUX(69);
```

3.5 ADCON register - Address: 2

The ADCON register has three main roles. By modifying bit 6-5 we can set up the clock out rate for the D0 pin of the ADS1256 chip. Modifying bit 4-3 sets up the sensor detect circuitry. Finally, bit 2-0 can change the PGA settings. For average users, these last three bits (bit 2-0) are the most relevant and most important bits. The rest is for more advanced applications.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	CLK1	CLK0	SDCS1	SDCS0	PGA2	PGA1	PGA0

Table 7: ADCON register bits. This register controls the PGA.

Bit 7 is a read-only bit, so we omit it. Bit 6-5 sets the CLKOUT (D0) pin. The value *00* turns the clock out OFF, which is the recommended usage. The value *01* makes the output to provide the f_{clkin} frequency on D0, *10* results in $f_{\text{clkin}}/2$ and *11* results in $f_{\text{clkin}}/4$ on D0. Bit 4-3³ sets the sensor detect current sources. The value *00* means it is OFF. The value *01*; means that the sensor detect current is set to $0.5 \mu\text{A}$. When it is set to *10*, the current is $2 \mu\text{A}$ and when it is *11*, the current is $10 \mu\text{A}$. The last three bits, bit 2-0 are responsible for the PGA setting.

PGA	Measurement range (V)	Bit 2-0
1	± 5	000
2	± 2.5	001
4	± 1.25	010
8	± 0.625	011
16	± 0.3125	100
32	± 0.15625	101
64	± 0.078125	110
64	± 0.078125	111

Table 8: PGA and register values with the corresponding measurement ranges.

³There's a typo in the original datasheet. The original datasheet says "Bits 4-2", which is wrong.

Since this register controls three different things: CLK, SDCS and PGA, three routines belong to this register.

```
1 void ADS1256::setCLKOUT(uint8_t clkout)
2 {
3     _ADCON = readRegister(ADCON_REG);
4
5     if(clkout == 0)
6     {
7         bitWrite(_ADCON, 6, 0);
8         bitWrite(_ADCON, 5, 0);
9     }
10    else if(clkout == 1)
11    {
12        bitWrite(_ADCON, 6, 0);
13        bitWrite(_ADCON, 5, 1);
14    }
15    else if(clkout == 2)
16    {
17        bitWrite(_ADCON, 6, 1);
18        bitWrite(_ADCON, 5, 0);
19    }
20    else if(clkout == 3)
21    {
22        bitWrite(_ADCON, 6, 1);
23        bitWrite(_ADCON, 5, 1);
24    }
25    else{}
26
27    writeRegister(ADCON_REG, _ADCON);
28    delay(100);
29 }
```

The above routine changes the bit 6-5 (CLK1 and CLK0) in the ADCON register. In the Arduino code it should be called in the following way:

```
1 setCLKOUT(0);
```

The above line will set the CLKOUT part of the ADCON register to *00* which will turn the clock out OFF. Further values can be 1 (*01*), 2 (*10*) and 3 (*11*).

The next part of the register to be written is the SDCS part which also has four values: 1 (00), 2 (01), 3 (10) and 4 (11).

```
1 void ADS1256::setSDCS(uint8_t sdcs)
2 {
3     _ADCON = readRegister(ADCON_REG);
4
5     if(sdcs == 0)
6     {
7         bitWrite(_ADCON, 4, 0);
8         bitWrite(_ADCON, 3, 0);
9     }
10    else if(sdcs == 1)
11    {
12        bitWrite(_ADCON, 4, 0);
13        bitWrite(_ADCON, 3, 1);
14    }
15    else if(sdcs == 2)
16    {
17        bitWrite(_ADCON, 4, 1);
18        bitWrite(_ADCON, 3, 0);
19    }
20    else if(sdcs == 3)
21    {
22        bitWrite(_ADCON, 4, 1);
23        bitWrite(_ADCON, 3, 1);
24    }
25    else{}
26
27    writeRegister(ADCON_REG, _ADCON);
28    delay(100);
29 }
```

The register is updated via the following command which in this example sets the corresponding part of the ADCON register to 00:

```
1 setSDCS(0);
```

Finally, the last part of the ADCON register is about the PGA setting. The PGA values and the corresponding effects are listed in [Table 1](#).

This code has been reiterated since the last update because it had a bug which messed with the rest of the `_ADCON`-related things.

First, we read the most recent value of the `ADCON` register and pass it to the `_ADCON` variable. Then, the left hand side operation (`_ADCON & 0b11111000`) clears all the PGA bits, and the right hand side operation (`pga & 0b00000111`) sets all the PGA bits while leaving the rest of the bits untouched. Then these two results are combined together using the logical OR operation. The result is sent to the `ADCON` register. Once the PGA is updated, the multiplier factor of the ADC conversion is also updated.

```
1 void ADS1256::setPGA(uint8_t pga)
2 {
3     _PGA = pga;
4     _ADCON = readRegister(ADCON_REG);
5     _ADCON = (_ADCON & 0b11111000) | (pga & 0b00000111);
6
7     writeRegister(ADCON_REG, _ADCON);
8     delay(200);
9
10    updateConversionParameter();
11 }
```

The function is used on the following way:

```
1 setPGA(PGA_8);
```

The above line will set the PGA value to `PGA_8` which corresponds to ± 625 mV input voltage range.

There's also a function to get the actual PGA value. This function simply reads the `ADCON` register and it returns only the PGA value by masking out the rest of the content of the register.

```
1 uint8_t ADS1256::getPGA() //Reading PGA from the ADCON register
2 {
3     uint8_t pgaValue = readRegister(ADCON_REG) & 0b00000111;
4     return(pgaValue);
5 }
```

Since the function returns a simple decimal number, it is a good idea to print its return value on the serial port as it is shown in the example below:

```
1 Serial.println(getPGA());
```

3.6 DRATE register - Address: 3

This register controls the sampling rate between 2.5 SPS up to 30000 SPS. The data rates and the corresponding register values are shown in Table 2. Keep in mind, that these data rates are valid for a single-channel acquisition. When multiplexing (switching between inputs) is used, the data rates are different due to the time needed for switching between the inputs.

I defined simple variables to call a certain data rate from the code:

```
1  #define DRATE_30000SPS 0b11110000
2  #define DRATE_15000SPS 0b11100000
3  #define DRATE_7500SPS 0b11010000
4  #define DRATE_3750SPS 0b11000000
5  #define DRATE_2000SPS 0b10110000
6  #define DRATE_1000SPS 0b10100001
7  #define DRATE_500SPS 0b10010010
8  #define DRATE_100SPS 0b10000010
9  #define DRATE_60SPS 0b01110010
10 #define DRATE_50SPS 0b01100011
11 #define DRATE_30SPS 0b01010011
12 #define DRATE_25SPS 0b01000011
13 #define DRATE_15SPS 0b00110011
14 #define DRATE_10SPS 0b00100011
15 #define DRATE_5SPS 0b00010011
16 #define DRATE_2SPS 0b00000011
```

The function that takes care of setting the data rate looks like this:

```
1 void ADS1256::setDRATE(uint8_t drate)
2 {
3     _DRATE = drate;
4     writeRegister(DRATE_REG, _DRATE);
5     delay(200);
6 }
```

The value of the data rate (DRATE) is received by the function as the argument. This value and the address of the DRATE register (DRATE_REG) is passed to the *writeRegister()* function. Then, after waiting for 200 ms, the function exits without a return value.

To apply a sampling rate, we only need to pass one of the above data rate variables to the *setDRATE()* function as it is shown below:

```
1 setDRATE(DRATE_60SPS);
```

The above function sets the data rate to 60 SPS.

3.7 I/O register - Address: 4

This register is a bit tricky, but it is not a big deal to understand it. Bit 7-4 sets the direction of the four I/O pins (D0-D3). Then, bit 3-0 can be both read and written. If they are read, the return value will reflect whether they are configured as an input (*1*) or as an output (*0*). If any of these pins are configured as an output, writing a value to them will change their state. If they are configured as input, writing them does not have an effect. Furthermore, the D0 pin has a special role because it also acts as the CLKOUT pin. So, it can be used either as an I/O pin or it can also be used to output a certain clock signal.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIR3	DIR2	DIR1	DIR0	DIO3	DIO2	DIO1	DIO0

Table 9: I/O register bits

There are three routines implemented for this register. One of them sets the I/O pins, another writes a value on them, and a third one reads them. It is important that first the GPIO pins have to be set as inputs or outputs and then we can write to them if applicable.

To set the GPIO pins of the ADS1256 as input or output, the bit 7-4 part is modified in the I/O register. To set the value we can call the *setGPIO()* function as it is shown below:

```
1 setGPIO(0,1,0,0);
```

The above line sets bit 4 to *0*, bit 5 to *1*, bit 6 and bit 7 to *0*. This means that D1 (bit 5 or DIR1) becomes an input and the other pins (DIR0, DIR2 and DIR3) become outputs. Please notice that the order of the parameters is according to the numerical order of DIR bits: the first parameter is DIR0 (bit 4), which is then followed by DIR1 (bit 5), DIR2 (bit6) and DIR3 (bit7).

If any of the pins are defined as outputs we can set the value of those specific pins. Let's continue the previous example where D0, D2 and D3 were set as outputs. In this example, DIO0 (bit 0) and DIO1 (bit 1) will be set to *1* which means that their state will be HIGH. The other two pins DIO3 (bit 3) and DIO2 (bit 2) will be set to *0*. The command will look like this:

```
1 writeGPIO(1,1,0,0);
```

The above line means that bit 0 (DIO0) and bit 1 (DIO1) of the I/O register become *1*, or HIGH as an output pin. Since bit 5 (DIR1) was set as an input previously, setting it to HIGH has no effect. Bit 2 and bit 3 are set to *0*, so DIR2 and DIR3 pins will set to LOW as outputs.

Finally, we can read the values of D0-D3 pins if they are set as inputs. This is done by the following command:

```
1 readGPIO(2);
```

In the above example, the bit 2 (DIO2) is read and the function returns with the state (HIGH or LOW) of the pin. This function reads only a single pin at a time. Another way to read the pin states at once is to read the whole register and read the first four bits:

```
1 readRegister(IO_REG);
```

Of course, the above example only works if the corresponding bits are set as inputs.

The *setGPIO()* function sets the four I/O pins as inputs (1) or outputs (0). The function takes four arguments; the first argument sets the D0 and the fourth argument sets the D3 pin.

```
1 void ADS1256::setGPIO(uint8_t dir0, uint8_t dir1, uint8_t dir2, uint8_t dir3)
2 {
3     _GPIO = readRegister(IO_REG);
4     uint8_t GPIO_bit7, GPIO_bit6, GPIO_bit5, GPIO_bit4;
5     if(dir3 == 1)
6     {
7         GPIO_bit7 = 1;
8     }
9     else
10    {
11        GPIO_bit7 = 0;
12    }
13    bitWrite(_GPIO, 7, GPIO_bit7);
14    if(dir2 == 1)
15    {
16        GPIO_bit6 = 1;
17    }
18    else
19    {
20        GPIO_bit6 = 0;
21    }
22    bitWrite(_GPIO, 6, GPIO_bit6);
23    if(dir1 == 1)
24    {
25        GPIO_bit5 = 1;
26    }
27    else
28    {
29        GPIO_bit5 = 0;
30    }
31    bitWrite(_GPIO, 5, GPIO_bit5);
32    if(dir0 == 1)
33    {
34        GPIO_bit4 = 1;
35    }
36    else
37    {
38        GPIO_bit4 = 0;
39    }
40    bitWrite(_GPIO, 4, GPIO_bit4);
41    writeRegister(IO_REG, _GPIO);
42    delay(100);
43 }
```

The *writeGPIO()* function takes four arguments. These are the logical values of the four pins between *0* and *1*. These values are only valid if the pins are set as outputs using the *setGPIO()* function, otherwise the values have no effect.

```
1 void ADS1256::writeGPIO(uint8_t dir0value, uint8_t dir1value, uint8_t dir2value,
2 uint8_t dir3value)
3 {
4     _GPIO = readRegister(IO_REG);
5     uint8_t GPIO_bit3, GPIO_bit2, GPIO_bit1, GPIO_bit0;
6     if(dir3value == 1)
7     {
8         GPIO_bit3 = 1;
9     }
10    else
11    {
12        GPIO_bit3 = 0;
13    }
14    bitWrite(_GPIO, 3, GPIO_bit3);
15    if(dir2value == 1)
16    {
17        GPIO_bit2 = 1;
18    }
19    else
20    {
21        GPIO_bit2 = 0;
22    }
23    bitWrite(_GPIO, 2, GPIO_bit2);
24    if(dir1value == 1)
25    {
26        GPIO_bit1 = 1;
27    }
28    else
29    {
30        GPIO_bit1 = 0;
31    }
32    bitWrite(_GPIO, 1, GPIO_bit1);
33    if(dir0value == 1)
34    {
35        GPIO_bit0 = 1;
36    }
37    else
38    {
39        GPIO_bit0 = 0;
40    }
41    bitWrite(_GPIO, 0, GPIO_bit0);
42    writeRegister(IO_REG, _GPIO);
43    delay(100);
44 }
```

The *readGPIO()* function reads the whole I/O register but only returns the value of the pin (or bit) requested via the argument of the function.

```
1  int ADS1256::readGPIO(uint8_t gpioPin)
2  {
3      uint8_t GPIO_bit3, GPIO_bit2, GPIO_bit1, GPIO_bit0;
4
5      _GPIO = readRegister(IO_REG);
6
7      GPIO_bit3 = bitRead(_GPIO, 3);
8      GPIO_bit2 = bitRead(_GPIO, 2);
9      GPIO_bit1 = bitRead(_GPIO, 1);
10     GPIO_bit0 = bitRead(_GPIO, 0);
11     delay(100);
12
13     if(gpioPin == 0)
14     {
15         return(GPIO_bit0);
16     }
17     if(gpioPin == 1)
18     {
19         return(GPIO_bit1);
20     }
21     if(gpioPin == 2)
22     {
23         return(GPIO_bit2);
24     }
25     if(gpioPin == 3)
26     {
27         return(GPIO_bit3);
28     }
29 }
```

3.8 Calibration registers - Address 5-10

This section discusses a set of registers. There are two types of registers related to the calibration, but there are six actual registers, three for each type of calibration. One type is the offset-calibration register (OFC, address: 5, 6 and 7) and another is the full-scale calibration register (FSC, address: 8, 9 and 10). The onboard calibration circuitry of the ADS1256 helps to minimize the offset and gain errors. The two previously mentioned registers OFC and FSC are actually made up of three 8-bit registers, so in total both OFC and FSC registers are 24-bit registers.

The output after calibration can be described with the following equation:

$$\text{Output} = \left(\frac{\text{PGA} \cdot V_{in}}{2 \cdot V_{\text{ref}}} - \frac{\text{OFC}}{\alpha} \right) \cdot \text{FSC} \cdot \beta, \quad (4)$$

where PGA is the value of the PGA (1-64), V_{in} is the input voltage, V_{ref} is the value of the voltage reference, OFC is the value of the offset calibration register, FSC is the value of the full-scale calibration register, and α and β vary with DRATE[1].

These registers can be manipulated by five different commands:

- SELFCAL – Self offset and self gain calibration
- SELFOCAL – Self offset calibration
- SELFGCAL – Self gain calibration
- SYSOCAL – System offset calibration
- SYSGCAL – System gain calibration

SELFCAL first performs a self offset calibration (SELFOCAL), then a self gain calibration (SELEFGCAL). During self calibration, the inputs are disconnected from the signal source. The self calibration time is proportional to the actual DRATE setting. In the slowest case (DRATE = 2.5 SPS), the self calibration takes 1.2272 seconds.

The SELFCOCAL performs a self offset calibration. In this process, the inputs are disconnected from the signal source and they receive AVDD/2 voltage. The SELFGCAL performs a self gain calibration. The inputs are disconnected from the signal source like in the case of the offset calibration, but they are connected to VREFP and VREFN. Important notice that the buffer is still involved which means that if a higher than AVDD – 2 voltage is used as the reference voltage, the buffer must be turned OFF. If AVDD is 5 V and VREF is 2.5 V, this is not an issue, there is still half a Volt headroom.

The sytem calibration differs from the self calibration in a sense that it requires an external calibration signal. The SYSOCAL performs a system offset calibration where the user must supply a zero input **differential** signal, i.e. the input must be shorted. I highlighted the word differential in the previous sentence, because this method would not work in a unipolar (i.e. $A_x + \text{AINCOM}$) scenario, the input must be a differential input (i.e. $A_x + A_y$).

To not overcomplicate things, I only use the SELFCAL function. I send this command after initializing the ADS1256. Actually, all five calibration procedures can be performed via the command definitions, which are defined in the ads1256.h header file:

```
1  #define SELFCAL 0b11110000
2  #define SELFOCAL 0b11110001
3  #define SELFGCAL 0b11110010
4  #define SYSOCAL 0b11110011
5  #define SYSGCAL 0b11110100
```

To call one of the above commands, we just need to use it in the *sendDirectCommand()* function:

```
1  sendDirectCommand(SELFCAL);
```


4 Coding

In the previous sections I summarized most of the important functions and functionalities related to the ADS1256 24-bit ADC. In the following section and sub-sections I share and document each functions of the library.

The code should work on Arduino-compatible microcontrollers using the regular Arduino IDE. Just unpack the zip file into a folder called ADS1256 under the *libraries* folder of your Arduino folder. I tested the code on Arduino Uno and Nano, ATmega32U4, Teensy 4.0, STM32F103C8T6 and STM32F401CCU6, and ESP32-WROOM-32 (38 pin board) microcontrollers. Due to the native USB support, higher clock speeds and larger memory I recommend the ST microcontrollers or the Teensy 4.0.

4.1 Constructor

The library works with a parameterized constructor. The constructor expects 5 parameters: DRDY pin, RESET pin, SYNC/PDWN pin, CS pin and the value of the reference voltage (V_{ref}). Due to the construction of certain off-the-shelf ADS1256 boards, I made it possible to omit the RESET pin and the SYNC/PDWN pin. Sometimes these pins don't have any terminals on the PCB, or they can be simply tied to the positive supply voltage by the user and there's no need to switch them. If they are not used, their value should be zero (0), so the code will know that it does not need to use those pins. The rest of the pins on a commercial ADS1256 PCB are the SPI pins. Each MCU has its own pin layout, so check their datasheet to see which pins are used for SPI communication. These pins are the MOSI, MISO and SCK pins.

```
1  ADS1256::ADS1256(const byte DRDY_pin, const byte RESET_pin,
2  const byte SYNC_pin, const byte CS_pin, float VREF)
3  {
4      _DRDY_pin = DRDY_pin;
5      pinMode(_DRDY_pin, INPUT);
6
7      if(RESET_pin !=0)
8      {
9          pinMode(RESET_pin, OUTPUT);
10     }
11     _RESET_pin = RESET_pin;
12
13     if(SYNC_pin != 0)
14     {
15         pinMode(SYNC_pin, OUTPUT);
16     }
17     _SYNC_pin = SYNC_pin;
18
19     _CS_pin = CS_pin;
20     pinMode(CS_pin, OUTPUT);
21
22     _VREF = VREF;
23     _PGA = 0;
24
25     updateConversionParameter();
26 }
```

Keep in mind that if you don't take care of the RESET and SYNC/PDWN pins either in the code or by connecting them to the positive voltage line, the ADS1256 will not communicate properly with the microcontroller! The easiest way to notice this mistake is that you cannot write the registers and when you read them, they return with 0 value; or when you try to read an acquisition, the code hangs.

The constructor looks quite simple. It passes the parameters to the private variables if applicable as well as it defines the pins as inputs or outputs. Finally, the value of the reference is saved in a private variable.

To create an instance, one can do as follows:

```
1  ADS1256 A(8, 10, 5, 9, 2.500);
```

The above line will create an instance called "**A**", where the DRDY pin is pin 8, the RESET pin is pin 10, the SYNC/PDWN pin is pin 5, the CS pin is pin 9 and the reference voltage is 2.500 V. The pins have to be chosen carefully to not interfere with the fixed SPI pins of the selected microcontroller or with other pins which might be used by other functions (TX/RX pins, or i2c pins...etc.).

Then, each function from the library can be called through this instance. For example, the DRATE can be changed in the following way:

```
1  A.setDRATE(DRATE_500SPS);
```

The above line tells the ADS1256 to change the sampling rate to 500 SPS.

To make the creation of the instance more clear, here is another example:

```
1  ADS1256 ADS1256_ADC(8, 10, 5, 9, 2.500);
```

The above line created an instance called "**ADS1256_ADC**", and you can call the functions like this

```
1  ADS1256_ADC.setDRATE(DRATE_500SPS);
```

4.2 Handling the DRDY signal

The code heavily relies on the DRDY signal since this signal tells the MCU when a conversion is ready and it can be fetched. Since this signal can change several ten thousand times per second (30 ksps!), depending on the selected sample rate, it is crucial to catch the changes properly, and most importantly, catch **all** of them!

Earlier, I implemented an interrupt-based handling of the DRDY pin, however, after discovering some issues with certain functions, I switched to a polling-based approach. However, after some testing, especially on faster microcontrollers, and user-feedback, it turned out that in some special cases, the polling-based approach can cause the *readSingleContinuous()* function to run multiple times between two DRDY pulses. For example, at 30 ksps sampling rate, the benchmark could return 49963 sps data rate. To avoid this issue, the latest update (v1.4) also contains a function that waits for the DRDY to go HIGH.

```
1 void ADS1256::waitForLowDRDY()
2 {
3     while (digitalRead(_DRDY_pin) == HIGH) {}
4 }

1 void ADS1256::waitForHighDRDY()
2 {
3     while (digitalRead(_DRDY_pin) == LOW) {}
4 }
```

The *waitForLowDRDY()* function is placed in the code where the MCU needs to wait for the DRDY signal to go LOW. The code waits in a while() loop until the DRDY pin changes to LOW. Once the pin is LOW, the rest of the code can proceed.

The *waitForHighDRDY()* function is used in the *readSingleContinuous()* function after the conversion to make sure that the DRDY returns to HIGH before the code starts to wait for it to go LOW again.

It might worth noticing that in the previous versions (v1.1 and below), there was a bug most probably caused by the interrupt-based DRDY handling. The DRDY pin was not treated properly and its status was not represented correctly throughout the library. This lead to a funny behaviour when reading the differential channels at high sampling rates. Sometimes, instead of real conversion values, values of the adjacent channels appeared on the output.

4.3 Initialization

After creating an instance, an initialization has to be done to set up the MCU and the ADC. This is done by calling the *InitializeADC()* function:

```
1 A.InitializeADC();
```

The function pulls CS low to indicate that we want to communicate with the ADC via the SPI. Then, if applicable a manual reset is performed by pulling the RESET pin LOW and keeping it LOW for at least t_{16} . This resets the whole ADC except CLK0 and CLK1 bits in the ADCON register. After releasing from the RESET, a self-calibration is automatically performed by the ADC. Then, if applicable, the SYNC pin is set to HIGH.

After resetting the ADC, the SPI and the interrupt are initialized. Then, most of the registers get some initial values to prepare the ADC for conversions. The last command is another self-calibration. Finally, there is a variable called *_isAcquisitionRunning* that keeps track of the acquisition. Certain parts of the library should only run once when the sampling is started and this variable ensures that the code uses or skips certain parts of the code when it is needed.

```

1 void ADS1256::InitializeADC()
2 {
3     digitalWrite(_CS_pin, LOW);
4
5     if(_RESET_pin != 0)
6     {
7         digitalWrite(_RESET_pin, LOW);
8         delay(200);
9         digitalWrite(_RESET_pin, HIGH);
10        delay(1000);
11    }
12
13    if(_SYNC_pin != 0)
14    {
15        digitalWrite(_SYNC_pin, HIGH);
16    }
17    SPI.begin();
18    attachInterrupt(digitalPinToInterrupt(_DRDY_pin), DRDY_ISR, FALLING);
19    delay(200);
20
21    _STATUS = 0b00110110;
22    writeRegister(STATUS_REG, _STATUS);
23    delay(200);
24
25    _MUX = 0b00000001;
26    writeRegister(MUX_REG, _MUX);
27    delay(200);
28
29    _ADCON = 0b00000000;
30    writeRegister(ADCON_REG, _ADCON);
31    delay(200);
32
33    _DRATE = 0b10000010; //100SPS
34    writeRegister(DRATE_REG, _DRATE);
35    delay(200);
36
37    sendDirectCommand(0b11110000);
38    delay(200);
39
40    _isAcquisitionRunning = false;
41 }

```

4.4 Converting the ADC count into a proper format

The following macro **has been introduced** to convert the 24-bit ADC reading into a correct signed 32-bit representation.

```
1  convertSigned24BitToLong(value) ((value) & (11 << 23) ? (value) - 0x1000000 : value)
```

The *value* is the ADC reading. The macro checks if the 24th bit of the *value* is set. If it is set, the macro subtracts 0x1000000 (16777216 or (2^{24})) from the acquired number, otherwise it leaves the *value* unchanged.

4.5 Converting the acquired value into voltage

Before the other functions that do the ADC conversion in different ways are described, an auxiliary function is introduced which helps us to convert the ADC readings into voltage values in a form of floating-point numbers. This function is called *convertToVoltage()*. The function requires a parameter, a 32-bit integer⁴ and it returns a floating-point number which is the measured voltage on the active input of the ADS1256. The previous paragraph (4.4) prepares this 32-bit integer already, this function only has to convert it according to the formula in the datasheet[1].

To simplify the operations in the function during continuous acquisition and conversion, a helper-function *updateConversionParameter()* has been introduced. Since the ADC conversion is just a multiplier away from being a voltage value, it is enough to calculate this multiplier once, for example when the user updates the PGA value, and then use it later when needed.

```
1  void ADS1256::updateConversionParameter()
2  {
3      conversionParameter = ((2.0 * _VREF) / 8388608.0) / (pow(2, _PGA));
4  }

1  float ADS1256::convertToVoltage(int32_t rawData)
2  {
3      return(conversionParameter * rawData);
4  }
```

A typical usage of this function looks like this:

```
1  float outputVoltage = 0;
2  outputVoltage = convertToVoltage(readSingle());
```

In the above example, we first created a floating-point variable (*outputVoltage*) that stores the conversion value. Then, we read a single conversion using the *readSingle()* function and we pass its return value to the *convertToVoltage()* function. Thus, the *outputVoltage* variable contains the conversion in Volts.

⁴Of course, the conversion result will be only 24 bits, but there's no such type as "int24_t". The 24-bit conversion result is stored in a 32-bit variable. The 8 remaining bits (24-31) are zero and only 0-23 are occupied by the actual data.

4.6 Reading a single conversion

One of the simplest conversion types is reading a single conversion using the *readSingle()* function. The function does not have any parameters and it returns the raw 24-bit value of the conversion. This function reads from the inputs selected by using the *setMUX()* function prior to calling this function. The return value is either converted into a voltage value using the *convertToVoltage()* function from the previous section or it is sent to a computer where further processing of the raw data is done.

```
1 long ADS1256::readSingle()
2 {
3     SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
4     digitalWrite(_CS_pin, LOW);
5     waitForLowDRDY();
6     SPI.transfer(B00000001);
7     delayMicroseconds(7);
8
9     _outputBuffer[0] = SPI.transfer(0);
10    _outputBuffer[1] = SPI.transfer(0);
11    _outputBuffer[2] = SPI.transfer(0);
12    _outputValue =
13    ((long)_outputBuffer[0]<<16) | ((long)_outputBuffer[1]<<8) | (_outputBuffer[2]);
14
15    digitalWrite(_CS_pin, HIGH);
16    SPI.endTransaction();
17
18    return(_outputValue);
19 }
```

The function works in the following way:

- 1.) The SPI communication is initiated and the chip select pin is pulled to low to indicate the beginning of the SPI transaction
- 2.) The code is waiting for the DRDY pin to go LOW. It checks the status of the DRDY pin by polling it using the *digitalRead()* function
- 3.) The code sends the RDATA command to the ADC after DRDY went LOW and it waits 7 μ s
- 4.) The three bytes are shifted out from the ADS1256 by sending dummy bytes (0) to it and the values are stored
- 5.) The three stored bytes that contain the conversion are put together into a single 24-bit number
- 6.) The chip select pin is set to HIGH and the SPI transaction is ended, the conversion is finished
- 7.) The function returns the 24-bit raw conversion value

The *readSingle()* function can be repeated indefinitely. For example, it can be put in a *for()* or a *while()* loop. It can be called any time when a single conversion is needed. The *readSingle()* function always uses a single input defined by the user using the *setMUX()* function. If there is a need for continuous conversion and/or multiplexing the inputs, the *readSingleContinuous()*, *cycleSingle()*, and *cycleDifferential()* functions are recommended.

4.7 Reading a single input continuously

This function reads a previously selected input channel continuously using the RDATAAC command. This function is called *readSingleContinuous()*. It does not have any parameters and it returns a conversion value when it is called and a conversion is finished on the ADS1256.

```
1 long ADS1256::readSingleContinuous()
2 {
3     if(_isAcquisitionRunning == false)
4     {
5         _isAcquisitionRunning = true;
6         SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
7         digitalWrite(_CS_pin, LOW);
8         waitForLowDRDY();
9         SPI.transfer(B000000011);
10        delayMicroseconds(7);
11    }
12    else
13    {
14        waitForLowDRDY();
15    }
16
17    _outputBuffer[0] = SPI.transfer(0);
18    _outputBuffer[1] = SPI.transfer(0);
19    _outputBuffer[2] = SPI.transfer(0);
20
21    _outputValue = ((long)_outputBuffer[0]<<16) | ((long)_outputBuffer[1]<<8) |
22    (_outputBuffer[2]);
23
24    _outputValue = convertSigned24BitToLong(_outputValue);
25
26    if(digitalRead(_DRDY_pin) == LOW) {waitForHighDRDY();}
27
28    return _outputValue;
29 }
```

When the function is called for the first time, the *_isAcquisitionRunning* variable is false, which allows the code to perform some setup, such as starting the SPI, pulling the chip select pin LOW and sending the RDATAAC command to the ADC. At the same time the *_isAcquisitionRunning* variable is set to true, so when the code enters this function again, the above listed processed are not performed again. Then, the code waits for a full DRDY level transition in the *waitForLowDRDY()* function. This function will "hang" the code until an interrupt (change of the DRDY pin) releases it. Finally, the conversion value is being captured and assembled similarly as in the *readSingle()* function. Due to the previously experienced strange conversion rates on high-speed microcontrollers, I added a line with the *waitForHighDRDY()* function after the ADC conversion is received to make sure that the DRDY line returns to HIGH before the code starts the next iteration. This ensures that only one conversion is performed between two DRDY pulses.

The *readSingleContinuous()* function must be called continuously which means that it must be placed in a *for()* or a *while()* loop:

```
1 long rawValue = 0;
2 for (long i = 0; i < 90000; i++)
3 {
4     rawValue = A.readSingleContinuous();
5 }
```

For example, the above code continuously reads the ADS1256 chip 90000 times. In each iteration, the return value of the *readSingleContinuous()* is passed to the *rawValue* variable. Of course, the variable is overwritten in each iteration, this is just an example of showing that the return value of the function has to be passed to a variable or to a function [4.5] for further processing. For example, to convert the raw digital values into voltage.

Keep in mind, that calling the *readSingleContinuous()* function in a *for()* loop as above is not the same as calling the *readSingle()* function in the same loop. If we call the *readSingle()* function let's say 90000 times, then we get 90000 samples with some arbitrary (as fast as possible) timing between the samples, and that's all. However, with the *readSingleContinuous()* function, the samples return with proper timing according to the sampling rate set by the *setDRATE()* function. Also, after the above *for()* loop finishes the 90000 iterations, the ADC is still doing the conversions. The RDATAAC command must be terminated by sending the SDATAAC command to the ADC, and the SPI connection needs to be terminated as well.

4.8 Stopping continuous conversions

Now we know how to start a continuous conversion, but as it is mentioned in the previous section, in the case of the continuous conversions which were started by the RDATAAC command, we also need to stop them using the SDATAAC command. I wrote a very simple function for this which does not only stop the conversion, but it also stops the SPI transaction as well as pulls the chip select pin HIGH.

```
1 void ADS1256::stopConversion()
2 {
3     waitForLowDRDY();
4     SPI.transfer(B00001111);
5     digitalWrite(_CS_pin, HIGH);
6     SPI.endTransaction();
7     _isAcquisitionRunning = false;
8 }
```

The above function first waits for DRDY to go low, then it sends the SDATAAC command to the ADC, then it pulls the chip select pin to HIGH, then it stops the SPI transfer and finally resets the *_isAcquisitionRunning* variable's status to false, so the code is ready to start another continuous conversion.

As one can notice, there is no *SPI.beginTransaction()* call in this function. The reason is because the SPI communication was already started when the continuous acquisition was started. Since the acquisition is continuous, the SPI is not closed until the acquisition is stopped. So in this function, we just simply send a new command (SDATAAC) to the ADS1256 chip which will then stop the sampling. Then, the SPI can be stopped.

4.9 Multiplexing through the 8 single-ended inputs

This function implements the multiplexing through the eight single-ended inputs from A0 to A7. The *cycleSingle()* function also continuously converts the data, so after the code finished returning the value on the A7 input, it starts over from the A0 again. The continuous conversion has to be stopped using the *stopConversion()* function.

```
1 long ADS1256::cycleSingle()
2 {
3     if(_isAcquisitionRunning == false)
4     {
5         _isAcquisitionRunning = true;
6         _cycle = 0;
7         SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
8         SPI.transfer(0x50 | 1);
9         SPI.transfer(0x00);
10        SPI.transfer(SING_0);
11        digitalWrite(_CS_pin, HIGH);
12        delay(50);
13        digitalWrite(_CS_pin, LOW);
14    }
15    else{}
16
17    if(_cycle < 8)
18    {
19        _outputValue = 0;
20        waitForLowDRDY();
21        switch (_cycle)
22        {
23            case 0: //Channel 2
24                SPI.transfer(0x50 | 1);
25                SPI.transfer(0x00);
26                SPI.transfer(SING_1);
27                break;
28
29            case 1: //Channel 3
30                SPI.transfer(0x50 | 1);
31                SPI.transfer(0x00);
32                SPI.transfer(SING_2);
33                break;
34
35            case 2: //Channel 4
36                SPI.transfer(0x50 | 1);
37                SPI.transfer(0x00);
38                SPI.transfer(SING_3);
39                break;
40
41            case 3: //Channel 5
42                SPI.transfer(0x50 | 1);
43                SPI.transfer(0x00);
44                SPI.transfer(SING_4);
45                break;
```

```

46
47
48
49     case 4: //Channel 6
50         SPI.transfer(0x50 | 1);
51         SPI.transfer(0x00);
52         SPI.transfer(SING_5);
53         break;
54
55     case 5: //Channel 7
56         SPI.transfer(0x50 | 1);
57         SPI.transfer(0x00);
58         SPI.transfer(SING_6);
59         break;
60
61     case 6: //Channel 8
62         SPI.transfer(0x50 | 1);
63         SPI.transfer(0x00);
64         SPI.transfer(SING_7);
65         break;
66
67     case 7: //Channel 1
68         SPI.transfer(0x50 | 1);
69         SPI.transfer(0x00);
70         SPI.transfer(SING_0);
71         break;
72     }
73
74     SPI.transfer(B11111100);
75     delayMicroseconds(4);
76     SPI.transfer(B11111111);
77     SPI.transfer(B00000001);
78     delayMicroseconds(7);
79
80     _outputBuffer[0] = SPI.transfer(0x0F);
81     _outputBuffer[1] = SPI.transfer(0x0F);
82     _outputBuffer[2] = SPI.transfer(0x0F);
83     _outputValue =
84     ((long)_outputBuffer[0]<<16) | ((long)_outputBuffer[1]<<8) | (_outputBuffer[2]);
85
86     _cycle++;
87     if(_cycle == 8)
88     {
89         _cycle = 0;
90     }
91 }
92
93 return _outputValue;
94 }

```

This function does not differ too much from the *readSingleContinuous()* function. The main difference is that the MUX register is manipulated "on the go" during the conversion. First, the code checks if the *_isAcquisitionRunning* variable is false. When the code enters the function for the first time, the variable is false, therefore the value of the *_cycle* is set to 0. The SPI is started and the SING_0 input is pre-selected, then chip select pin is pulled to LOW. Then the code progresses and it waits for the DRDY to go LOW. Then, based on the value of the *_cycle* variable, an input is selected by writing the corresponding value to the MUX register. After this, a SYNC, then a WAKEUP command is sent to the ADC which is then followed by the RDATA command. Following a brief delay, the conversion value is shifted out and assembled into a 24-bit value. Then, the value of the *_cycle* variable is increased by one, it becomes 1 after the first iteration.

Since this function also runs indefinitely, the next iteration runs again, but now without the initialization of the SPI and the manipulation of the chip select pin. However, the value of the *_cycle* variable is now 1, so the code will select the SING_2 input. The function increments the *_cycle* up until 8. When it reaches 8 (which means that we just read the A7 pin), it is immediately reset to 0 after the conversion, so the multiplexing starts over from 0 (A0 pin) in the next iteration.

I think a convenient way to handle the continuous multiplexing of the 8 single-ended channel looks like this:

```

1  while (Serial.read() != 's')
2  {
3      float channels[8];
4
5      for (int j = 0; j < 8; j++)
6      {
7          channels[j] = A.convertToVoltage(A.cycleSingle());
8      }
9
10     for (int i = 0; i < 8; i++)
11     {
12         Serial.print(channels[i], 4);
13         if(i < 7)
14         {
15             Serial.print("\t");
16         }
17     }
18     Serial.println();
19 }
20 A.stopConversion();

```

The above code runs indefinitely until we send an *s* letter to the Arduino via the serial port. The code stores the 8 conversions in a buffer and after the 8th iteration, the code prints the numbers as a nice, formatted line:

```

1  2.1293    2.3457    1.2346    3.3345    1.2765    0.1245    0.0223    2.6712
2  0.7548    1.3587    2.4326    0.1234    3.1120    1.2901    2.3720    2.3475

```

Once the MCU receives the letter *s* from the serial port, it stops the conversion and ends the printing. It must be noted that with the above implementation, the code is only performing whatever is included in the while() loop. The code is "locked" inside the while() loop until the user send the *s* letter to the MCU.

4.10 Multiplexing through the 4 differential inputs

Essentially, the principles are the same as for the previous case for single-ended inputs. The main difference is that the code selects differential input channel pairs instead of single-ended ones. Thus, we only have 4 channels because the differential input channels are always created as a combination of two positive input terminals; for example A0 and A1.

```
1 long ADS1256::cycleDifferential()
2 {
3     if(!_isAcquisitionRunning == false)
4     {
5         _cycle = 0;
6         _isAcquisitionRunning = true;
7         SPI.beginTransaction(SPISettings(1920000, MSBFIRST, SPI_MODE1));
8         digitalWrite(_CS_pin, LOW);
9         SPI.transfer(0x50 | 1);
10        SPI.transfer(0x00);
11        SPI.transfer(DIFF_0_1);
12        digitalWrite(_CS_pin, HIGH);
13        delay(50);
14        digitalWrite(_CS_pin, LOW);
15    }
16    else
17    {}
18
19    if(_cycle < 4)
20    {
21        _outputValue = 0;
22        waitForLowDRDY();
23
24        switch (_cycle)
25        {
26            case 0:
27                SPI.transfer(0x50 | 1);
28                SPI.transfer(0x00);
29                SPI.transfer(DIFF_2_3);
30                break;
31
32            case 1:
33                SPI.transfer(0x50 | 1);
34                SPI.transfer(0x00);
35                SPI.transfer(DIFF_4_5);
36                break;
37
38            case 2:
39                SPI.transfer(0x50 | 1);
40                SPI.transfer(0x00);
41                SPI.transfer(DIFF_6_7);
42                break;
43    }
```

```

44     case 3:
45         SPI.transfer(0x50 | 1);
46         SPI.transfer(0x00);
47         SPI.transfer(DIFF_0_1);
48         break;
49     }
50
51     SPI.transfer(0b11111100);
52     delayMicroseconds(4);
53     SPI.transfer(0b11111111);
54     SPI.transfer(0b00000001);
55     delayMicroseconds(7);
56
57     _outputBuffer[0] = SPI.transfer(0);
58     _outputBuffer[1] = SPI.transfer(0);
59     _outputBuffer[2] = SPI.transfer(0);
60
61     _outputValue =
62     ((long)_outputBuffer[0]<<16) | ((long)_outputBuffer[1]<<8) | (_outputBuffer[2]);
63
64     _cycle++;
65     if(_cycle == 4)
66     {
67         _cycle = 0;
68     }
69 }
70 return _outputValue;
71 }

```

The first thing to do after entering the function is to check the status of the `_isAcquisitionRunning` variable. If it is false, we start the SPI communication, we set the first input channel to DIFF_0_1 (A0+A1 pins) and toggle the chip select pin. Then, the code waits for the DRDY pin to go LOW. Then, according to the value of the `_cycle` register an input pin pair is selected. Then a SYNC and a WAKEUP command is sent to the ADC, followed by the RDATA command. After a brief delay, the conversion is shifted out and assembled into a 24-bit value. The cycles repeat until the last input pin pair (A6+A7, `_cycle == 4`) is selected. Then after the conversion, the value of the `_cycle` variable is reset to zero. So, with the next iteration, the conversion will be obtained again from the A0+A1 input pin pair. Fetching and formatting the conversions can be done in the exact same way as for the `cycleSingle()` function:

```

1  while (Serial.read() != 's')
2  {
3      float channels[4];
4
5      for (int j = 0; j < 4; j++)
6      {
7          channels[j] = A.convertToVoltage(A.cycleDifferential());
8      }
9
10     for (int i = 0; i < 4; i++)
11     {
12         Serial.print(channels[i], 4);
13         if(i < 3)
14         {
15             Serial.print("\t");
16         }
17     }
18     Serial.println();
19 }
20 A.stopConversion();

```

The code stores the four conversions in a buffer and once the buffer is populated, the content of it is printed on the serial terminal as a formatted line:

```

1  2.1253    1.2356    3.0345    0.1845
2  0.7248    2.4306    2.3450    2.3405

```

5 Testing the code

In this section I show some test results with my library and my custom build ADS1256 voltage logger board. These are not extremely scientific tests, but they are good enough to show that the code and the ADC performs reasonably well. The idea is to show how precisely we can read the voltage of a 2.5 V voltage reference with and without the input buffer. Then I also want to show that the ADC can provide the expected sampling rates.

5.1 Precision

For these investigations I was using my AD584-based voltage reference and its 2.5 V output. To check its output voltage, I was using my 6.5-digit Solartron 7060 digital voltmeter [2]. Obviously, none of these are freshly calibrated, but they are used as my "in-house reference" for this investigation. All devices in the experiment were running for more than an our before doing the measurements. If a user needs the ADS1256 for more accurate studies, a proper calibration is recommended.

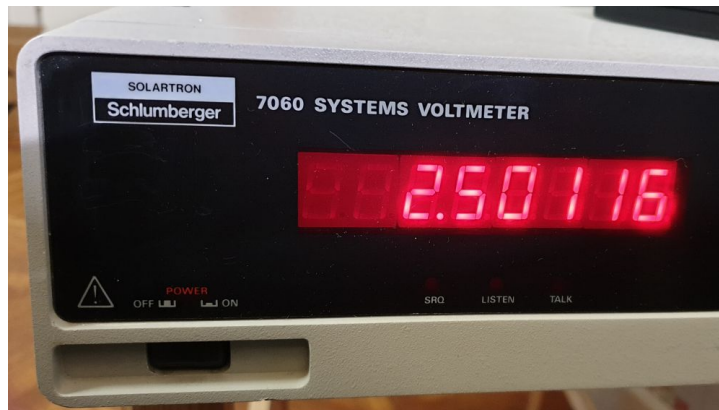


Figure 2: Picture of the value measured by my Solartron 7060 multimeter.

5.1.1 With input buffer

In this test, I turned the input buffer on and did 100 measurements at 100 SPS and took the average of the samples. The PGA was set to default (0).

According to my multimeter, the reference voltage was 2.50116 V, and according to ADS1256 it was 2.49963 V. The difference in the readings is 0.00153 V or 1.53 mV.

5.1.2 Without input buffer

Turning the input buffer off causes a serious disturbance in the measured voltage. I repeated a similar experiment that I performed with the input buffer being on. The only difference is that I did a self-calibration using the SELFCAL command after changing the buffer.

The multimeter was still showing the same value, (2.50116 V) however, the average of the 100 readings gave 2.09241 V. This is nearly half a Volt difference, 0.40875 V, to be precise. I tried to play around with the different settings and calibration, but I was not able to get rid of this nearly half a Volt offset. Therefore, I believe that this significant difference is really caused by the input buffer. It makes sense as the input impedance is significantly higher with the enabled buffer (80 M Ω). With the disabled buffer the input impedance becomes 150 k Ω . This impedance can be low enough to cause a voltage drop and influence the results.

5.2 Sampling rate measurement

I made a simple script that tests my code and checks the sampling rate. The principle of the script is simple, I start a timer, then after a certain number of collected samples, I stop the timer, stop the conversion and based on the elapsed time and the number of acquired conversions I calculate the "real sampling rate".

```
1 long numberOfSamples = 150000;
2 long finishTime = 0;
3 long startTime = micros();
4
5 for (long i = 0; i < numberOfSamples; i++)
6 {
7   A.readSingleContinuous();
8 }
9
10 finishTime = micros() - startTime;
11
12 A.stopConversion();
13
14 Serial.print("Total conversion time for the samples: ");
15 Serial.print(finishTime);
16 Serial.println(" us");
17
18 Serial.print("Sampling rate: ");
19 Serial.print(numberOfSamples * (1000000.0 / finishTime), 3);
20 Serial.println(" SPS");
```

In the example code above, the variables are adjusted for 30 kSPS sampling speed. The code collects 150000 samples, so it is expected to finish in about 5 seconds. Longer sampling would probably improve the results to some extent as the sampling speed would be averaged for a longer time period. In [Table 10](#) I summarized my results. There is a slight undershoot at 30 kSPS nominal sampling rate that I could not figure out so far, but losing 200 SPS is not mission-critical for me, so I am not too worried about the discrepancy. However, I suspect that the 16 MHz microcontrollers might be a bit too slow for this task, or my code is not efficient enough. Testing the same code on an STM32F103C8T6 (72 MHz), on a Teensy 4.0 (600 MHz) and on an ESP32-WROOM-32 (240 MHz) gave results slightly above 30000 samples per second.

As a side note, I want to emphasize, that this test is only testing the microcontroller's capability of fetching the conversions from the ADS1256 via SPI. As you can see, I just have a `for()` loop that iterates the `readSingleContinuous()` function n -times. The data is not being transferred to the computer!

If you need to get those 30000 samples to the computer in real time, then keep in mind that you need to send 30000×32 bits (120 kilobytes) every second. The function `Serial.println()` will probably not be able to keep up with such speed. It is a better idea to send the raw 32-bit data using the `Serial.write()` function and then reconstruct the values on the receiving side. A fast and efficient data transfer to the computer, however, is outside of the scope of this paper.

Nominal DRATE (SPS)	Number of samples	Sampling length (μ s)	Measured DRATE (SPS)
30000	150000	5032096	29808.7
15000	75000	4999708	15000.9
7500	37500	4999640	7500.5
3750	18750	4999472	3750.4
2000	10000	4998844	2000.5
1000	5000	4998244	1000.4
500	2500	4996316	500.4
100	500	4980844	100.4
60	300	4967256	60.4
50	250	4961504	50.4
30	150	4934324	30.4
25	125	4922852	25.4
15	75	4868452	15.4
10	50	4806848	10.4
5	25	4774140	5.2
2.5	12.5	5026844	2.5

Table 10: Nominal and measured sampling rates on my ATmega32U4-based board.

The above results are visualized in Figure 3 below. The image shows a nice agreement between the nominal and measured sampling rates.

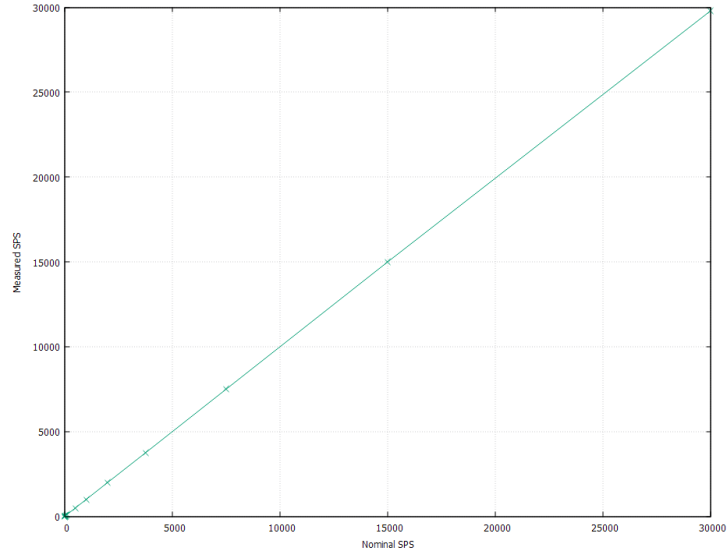


Figure 3: Nominal SPS vs measured SPS: Visualization of Table 10.

6 List of tested microcontrollers

In this section you can read the list of the microcontrollers I tested with the library.

- Arduino Uno
- Arduino Uno R4 WiFi⁵
- Arduino Nano
- Arduino Mega 2560
- ATmega32U4
- Teensy 4.0
- STM32F103C8T6
- STM32F401CCU6
- ESP32-WROOM-32 (38 pin board)
- RP2040 (Waveshare Zero, Pico W as well as my custom PCB)

⁵Thanks for [MaxMax-embedded](#) for testing and contribution!

7 Versions and changelog

7.1 First release - 2022-07-14

This is the first release of the library. Probably there are several mistakes in the code that I could not spot, but based on the feedbacks I hopefully will receive, these mistakes will be fixed. This handbook could also contain mistakes and errors, but they will be fixed in the upcoming versions as well.

7.2 Changed PayPal link - 2023-01-20

I had to change PayPal link due to administrative reasons, therefore a new document has been compiled.

7.3 Updates - 2023-11-10

Fixed a few typos in the text. Thank you for the feedbacks!

I also modified the text a little bit here and there to make it more consistent. I reworked the code to detect the DRDY pin change via an interrupt. Therefore I added a small section about this part of the code.

I also added a little explanation around the ESP32 chip. The library is now also compatible with the ESP32 chip after a little modification.

Thanks for Abraão Queiroz for testing the library and for recommending corrections!

Finally, the library is available on my [GitHub](#)! The code will be maintained there in the future so other programmers can play around with the library in a better environment.

7.4 Updates - 2024-02-08

A few bugs were fixed. Thank you for the feedbacks!

I fixed a bug regarding setting the PGA value. The PGA value is a part of the whole ADCON register, and when I modified the PGA value, I accidentally messed up the whole register's value. This should be fixed now.

Since the STATUS register has three important bits that can be set individually (byte order, auto-calibration and analog input buffer), I decided to also implement a "get function" for them which can be used to individually query the settings of these STATUS bits. It is a convenient shortcut.

Further "set functions" which only modify parts of the register value are modified in a way that we first fetch the most recent value of the register.

When the ADS1256 is initialized, we first pass the initial register values to the corresponding variables and then write these values on the registers, using the freshly assigned variables. This and the above fixes should solve a few more "not yet discovered" bugs. In some cases, the registers were updated without updating their values in their corresponding variables, however, other functions used these variables. This could lead to conflicting/confusing settings.

These new changes are published as v1.1. The update should be visible in the [Official Arduino Library Manager](#) as well.

7.5 Updates - 2024-09-12

An interesting bug was fixed.

With the help of [Benjamin Pelletier](#), an interesting bug was fixed. During cycling and reading the differential inputs using the *cycleDifferential()* function, there was a "crosstalk" between adjacent channels. This was visible as a repeated value of the n-th channel's conversion on the (n+1)-th channel. For example, if the 1st differential channel read 2.5433, sometimes the 2nd channel also returned 2.5433 instead of its real conversion result. Other adjacent channels could also suffer from this issue.

The fix involved changing the logic for reading the DRDY pin. From now on, the DRDY pin is directly read via polling instead of using interrupts.

I also updated the Arduino example code of the *cycleDifferential()* and *cycleSingle()* functions so they work in a buffered mode. First, the 4/8 channels are stored in a buffer and then before performing another set of 4/8 conversions, the buffer is sent to the PC via the serial port.

These new changes are published as v1.2. The update should be visible in the [Official Arduino Library Manager](#) as well.

7.6 Updates - 2025-03-28

Several improvements have been implemented in this patch.

To make the code more readable, I refractored it to some extent. In the *cycleSingle()* and *cycleDifferential()* functions I replaced the repetitive lines that switch the MUX to the new channel with a helper function. It does not change anything functionally, but it makes the code more readable.

I removed some unused variables and introduced local variables instead.

I removed all the `Serial.print()` functions from the library. This allows the users to format the outputs of the functions as they want. Consequently, certain functions now return with a number instead of a text message. The involved functions are:

- *getPGA()*
- *getByteOrder()*
- *getBuffer()*
- *getAutoCal()*
- *readGPIO()*

In the example .ino file the case 'P' that sets the PGA value had an error. The parsing line was before the line that waits for the established serial connection. I swapped the lines so now the code first waits for the serial port to become available and then parses the number coming through the serial port. I also added a line that prints the value of the PGA by reading the corresponding register.

With the help of a GitHub user [RadoMmm](#), the conversion from ADC reading to voltage has been optimized. Previously, the *convertToVoltage()* function performed a set of calculations for every single conversions. But actually it always calculated the exact same multiplication factor and then multiplied the ADC reading with this value. From now on, the multiplication factor (*conversionParameter*) is only updated when the user changes the PGA setting and then it is used as a multiplier in the *convertToVoltage()* function.

These new changes are published as v1.4. The update should be visible in the [Official Arduino Library Manager](#) as well.

References

- [1] <https://www.ti.com/product/ADS1256>
- [2] http://www.caddock.com/Online_catalog/voltagedividers_networks/voltageDividers_networks.html