ADS1015 12 bit ADC
Asynchronous library for Arduino

July 20, 2017
Ton Rutgers

12 or 11 bit?

A few weeks ago I bought an ADS1015 12bit ADC for my Arduino project. I needed more precision for two analog inputs than the standard 10bit ADC's from my Uno. When I got it, I examined the data sheet more careful, and found out it was not what I expected:

- I expected 4 independent ADC's. Although the ADS1015 has 4 input channels, it has one ADC. Each input channel can be connected internally to the input of the ADC, and then the ADC will take a sample of that input. Then you can switch to the next input and so on;
- And I expected 12 bit resolution from ground (0V) to Vdd (+5V power supply for Uno). The ADS1015 does not provide that. Let me explain. The ADS1015 can operate in two modes:
  - In <u>single ended mode</u> you can measure from ground to Vdd. In this mode the resolution is not 12, but 11 bit! You get a range from 0 … 2047, not 0 … 4095;
  - The 12 bit resolution can only be achieved in <u>differential mode</u>. In this case the ADS1015 measures the difference between two inputs. However, any input cannot be lower than ground and not higher than Vdd. Exceeding those limits may destroy the ADC. Let me give an example: you measure the difference between inputs AIN0 and AIN3. Suppose input AIN0 is 4.000V and input AIN3 is 1.000V. You will get a result of:
    AIN0– AIN3 = 4.000 – 1.000 = +3.000V. Now suppose AIN0 is 1.000V and AIN3 is 4.000V. Then you get: AIN0 – AIN3 = 1.000 – 4.000 = -3.000V, a negative voltage! This is how the ADS1015 works, you can get a result of (in digits) -2048 … +2047, which is a 12 bit resolution.

I think the practical value for 12 bit *differential* readings is limited for most Arduino projects. In my opinion +/-11 bit is a better description for the ADS1015. For my particular project I won't be able to use the 12 bit capability, but I *will* use the differential mode in 11 bit resolution. I will come back to this later.

Adafruit library

Adafruit supplies great libraries. So I tried the Adafruit library, which is (as far as I know) the only library for the ADS1015. This library also supports the ADS1115 ADC, which is 16 bit; to be more precise: +/- 15 bit. I did not understand this library very well, but that was probably caused by the fact I did not understand the ADS1015, yet. There also seems to be a bug in the library, because it reports the same result for all 4 single ended inputs, while there were 4 different voltages at each input. Same thing for the two differential modes: differential AIN0-AIN1 and differential AIN2-AIN3 gave the same result with different voltages at the inputs.
Another thing with the Adafruit library is that it implements only two differential modes:

- AIN0 -> AIN1
- AIN2 -> AIN3.

But these two are missing:

- AIN0 -> AIN3
- AIN1 -> AIN3.

And these particular modes I want to use in my project.
Finally, the Adafruit library does not check the ADS1015 configuration register if the conversion is completed. It simply waits 1ms before reading the result. This may be the bug in the library, because according to my measurements a conversion takes longer than 1ms, so the result is not available yet.

Conversion speed

The ADS1015 is not fast. A single conversion takes 1.65 ms. Compared to the 0.1 ms of the native 10 bit Uno ADC's, this is slow. The datasheet specifies 3300 for the highest rate for samples per second (SPS). This would mean a sample time of 1/3300 = 0.3 ms. The actual sample time of 1.65 ms doesn't come close. I think[1] this is because the datasheet specifies 3300 SPS (0.3 ms) for 'continuous mode', which means the chip is continuously sampling *one* of the inputs. So apparently reconfiguring the ADC to switch from one input to another, takes time: 1.65 – 0.3 = 1.35 ms.
During my research I noticed the performance is greatly influence by:

- Serial printing, especially at slow baud rates. For example 9600 baud;
- The I2C bus speed, which is standard at 100 kHz.

The library includes an example sketch, and you can turn on/off serial printing and switch between the standard I2C clock rate of 100kHz and fast mode, 400 kHz. The next table shows the influence on conversion time measurements:

Serial printing (baud)	I2C clock (kHz)	Conversion time (µs)
115200	100	3251
115200	400	2324
Off	100	2408
Off	400	1640

Asynchronous library

My project does a lot of real time stuff, and I don't want to wait 1.65 ms for each input, because that would be almost 7 ms for 4 inputs. Therefore I developed an asynchronous library for the ADS1015. With 'asynchronous' I mean my sketch can start a conversion, do other things and come back later to check if the ADC is ready and fetch the result. This is great for real time processing, because the sketch doesn't have to wait for the ADC to complete; the ADC does it's conversion in the background, without taking CPU time.
In this library I also implemented:

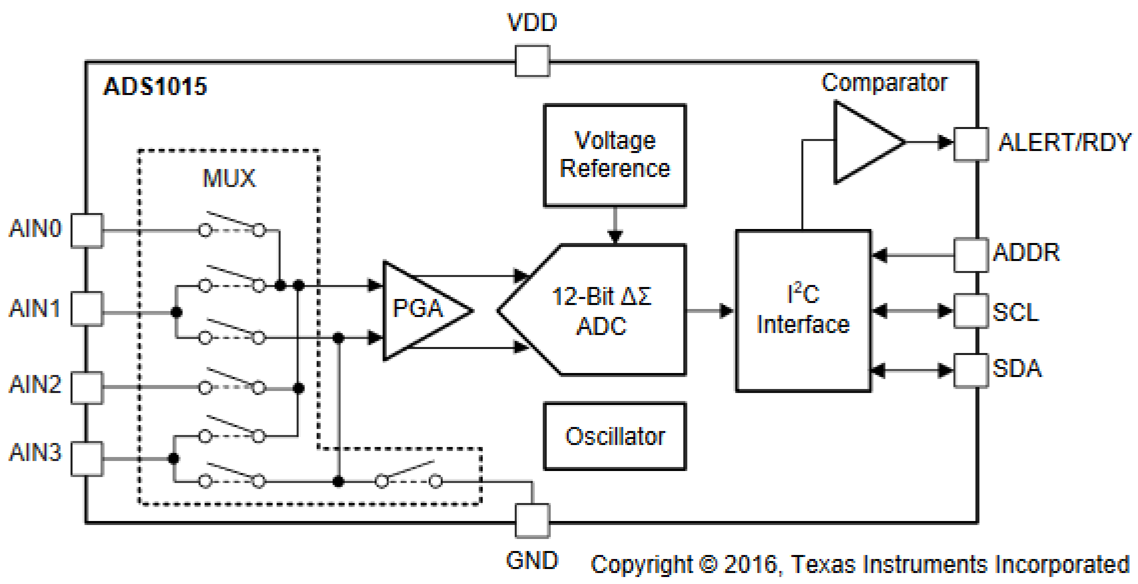
- ALL possible input combinations of the ADS1015. So the missing combinations in the Adafruit library AIN0 -> AIN3 and AIN1 -> AIN3 are also supported;
- Configurable automatic gain adjustment. For each input mode you can choose fixed gain or automatic adjustment.

I did not implement support for the comparator capability of the ADS1015, because I do not plan to use this.

A closer look at the ADS1015

Let's have a closer look at the ADS1015. The next figure is a functional diagram from the datasheet. I will discuss the components from left to right.

[1] The datasheet is not clear about this, so this is an assumption.



Power supply

The ADS1015 can operate at a power supply from 2.0 to 5.5V, so it suits for both 3.3V and 5.0V Arduino's. I have not tested it with 3.3V Arduino's, but I believe it should work.

Inputs

The ADS1015 has 4 inputs: AIN0, AIN1, AIN2 and AIN3. These inputs can handle a voltage from ground – 0.3V up to Vdd + 0.3V. Each input is protected for overvoltage. The chip will be destroyed if the input *current* exceeds 10mA. Therefore it is always a good idea to connect a resistor between the voltage to be measured and the input. With a series resistor of 1kΩ the voltage to be measured can go from -10 to +15V without harming the chip. I have tested this with a series resistor of 18kΩ, and the measurement went up to 5.6V when providing +9V and -0.12V when providing -9V.
Note: the input resistance is fairly high, around 3MΩ in most cases, but can go as low as 700kΩ in differential mode for the +/- 0.512V and +/-0.256V scales. This will introduce some error in the measurement if you connect a series resistor between the voltage to be measured and the input:

	Error (%)	
Input mode	Series resistor 10kΩ	Series resistor 1kΩ
Differential +/- 0.512V and +/-0.256V scales	-1.41%	-0.14%
All other scales	-0.33%	-0.03%

Multiplexer

The inputs connect to the multiplexer. It provides 8 possible input configurations:

- Differential with two measurement points:
    - 0: AIN0 -> AIN1
    - 1: AIN0 -> AIN3
    - 2: AIN1 -> AIN3
    - 3: AIN2 -> AIN3
- Single ended, the other measurement point is internally connected to ground:
    - 4: AIN0
    - 5: AIN1
    - 6: AIN2
    - 7: AIN3.

The multiplexer is controlled by writing a 3 bit number to the configuration register.

Programmable Gain Amplifier (PGA)

The PGA is a cool feature of the ADS1015. You can set the gain and change the sensitivity of the inputs. Possible scales are:

			Vdd = 5.0V		Vdd = 3.3V	
Scale#	Full Scale range	Gain (V/digit)	Actual scale range	Raw result (digits)	Actual scale range	Raw result (digits)
0	+/-6.144V	0.003000	+/-5.000V	-1666 … +1665	+/-3.300V	-1100 … +1099
1	+/-4.096V	0.002000	+/-4.096V	-2048 … +2047	+/-3.300V	-1650 … +1649
2	+/-2.048V	0.001000	+/-2.048V	-2048 … +2047	+/-2.048V	-2048 … +2047
3	+/-1.024V	0.000500	+/-1.024V	-2048 … +2047	+/-1.024V	-2048 … +2047
4	+/-0.512V	0.000250	+/-0.512V	-2048 … +2047	+/-0.512V	-2048 … +2047
5	+/-0.256V	0.000125	+/-0.512V	-2048 … +2047	+/-0.512V	-2048 … +2047

Notes:

- The limits in scale 0, +/- 6.144V, will never be reached, because the supply voltage is limiting;
- For 3.3V power supply this also applies to scale 1, +/- 4.096V.

The PGA can bet set by writing a 3 bit value to the configuration register.

ADC, voltage reference & oscillator

Well not much to tell about this. As described before it is a 12 bit ADC, but it would be better to call it a +/- 11 bit ADC.

I2C interface

The ADS1015 has an ADDR pin, with which you can configure the I2C address. You have to connect this pin to another pin to get the appropriate I2C address. If you leave the ADDR pin unconnected, the I2C address will be 0x48, but to make sure it is better to connect the pin:

ADDR pin connection	I2C address (hex)
GND	0x48
Vdd	0x49
SDA	0x4A
SCK	0x4B

You can specify the I2C address to the library.

The I2C interface can run at a clock signal of 100kHz (standard I2C) or in fast mode (400kHz). You can simply set the clock in your sketch to 400kHz with:

```
Wire.setClock(400000);
```

It is possible to communicate to the ADS1015with clock speeds up to 3.4MHz, but then it gets more complex and I have not figured that out.

Comparator

The ADS1015 contains a comparator that may be configured to trigger the ALERT/RDY output. You can generate an interrupt to the Arduino. I have not gone deeper into this, and configuring the comparator it is not implemented in the library.

Accuracy & noise

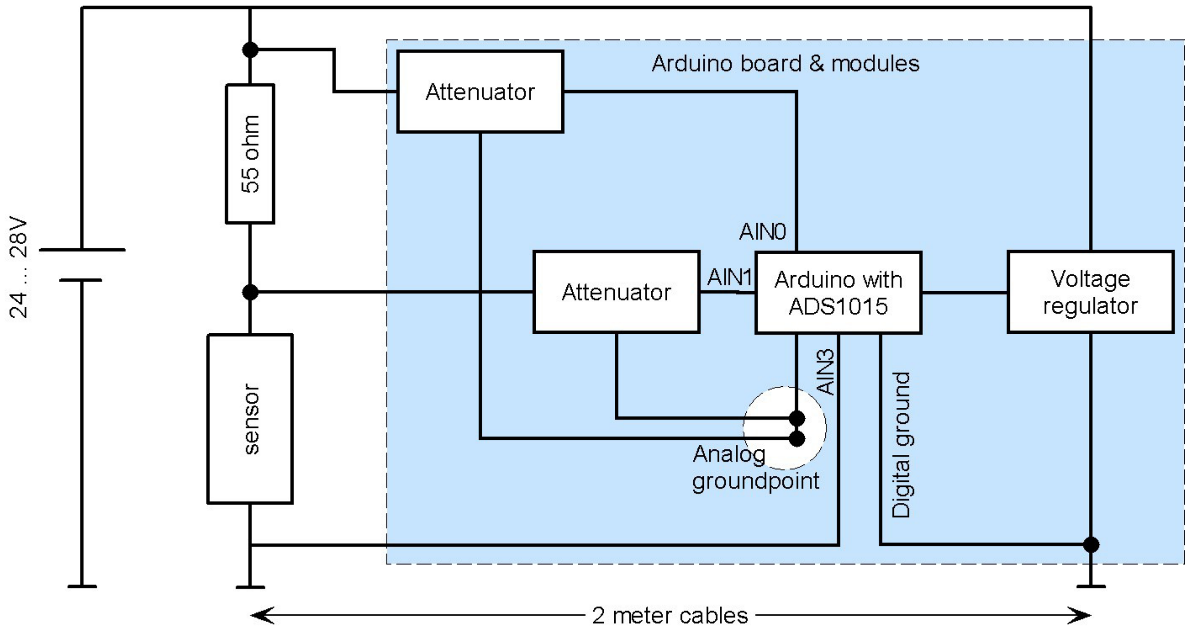
The ADS1015 is quite accurate: +/-0.05%. Which is approximately +/-1 least significant bit.

Noise may be a problem. This is not specific for the ADS1015, but a general problem for analog circuitry with high accuracy requirements. The ADS1015 provides a useful feature for reducing noise: the differential input modes. Noise that is present in both the ground wire and the wire to the measurement point, can be eliminated using the differential input modes. This is especially useful for 'noisy' environments and long cables. The trick is to use the differential modes:

- AIN0 -> AIN3
- AIN1 -> AIN3
- AIN2 -> AIN3.

AIN3 is used as reference point and AIN0, AIN1 and AIN2 will measure the voltage. For this you have to make the connections close to the object you want to measure. In most cases AIN3 will be at ground level, but as close to the sensor as possible. Any noise due to current in ground wires will be eliminated. This technique is quite common to avoid noise and hum from the power grid for amplification of small signal audio devices like microphones.

For my project I am using the next circuit:



The sensor is connected to ground quite far away from the Arduino and the ADS1015. In theory all grounds are at the same voltage level, but they are not! They *will* be different and it affects the measurements, especially in the mV range. By using differential mode by connecting AIN3 close to the sensor and AIN0 and AIN1 close to the other sides, ground noise is eliminated.

Other tips for reducing noise:

- Separate digital and analog ground by creating an analog-only ground point. This can be done by reserving one of the two Arduino ground pins for analog signals *only*;
- Shielded cables for long signal wires;
- Connect a 0,1µF capacitor between ground and power supply voltage of each digital breadboard/module connected to your Arduino;
- As last resort you can use software filtering with the formula: $y_n = (1 - \alpha) * y_{n-1} + \alpha * y_{new}$
  Where:

  y_{new} the new measured value

y_{n-1} is the previous *filtered* value
y_n is the new *filtered* value which includes the new measured value
α is the filter factor and is a number between 0 and 1. A low α will have a high damping effect and a high α will have low damping. Note: damping will slow down *change* in measurement.

How to use the asynchronous library

Configuration variables:

```
Byte I2Caddres = 0x48;
Byte inputSelect        =  B00000110; // select diff. AIN0-3 and AIN1-3
Byte autoGainAdjust     =  B00000110; // set auto gain for both inputs
Unsigned long inputGain = 0x00000310; // set maximum gain
```

Variable I2Caddress
The I2C address of the ADS1015; 0x48 is the default if you connect the ADDR pin to ground.

Variable inputSelect
Each bit in this variable will enable one of eight possible input modes. Bit 0 is the rightmost bit:

bit	description
0	enable differential AIN0 -> AIN1
1	enable differential AIN0 -> AIN3
2	enable differential AIN1 -> AIN3
3	enable differential AIN2 -> AIN3
4	enable AIN0 single ended
5	enable AIN1 single ended
6	enable AIN2 single ended
7	enable AIN3 single ended

There is no restriction in selecting input modes; *all* combinations are possible, so you can mix single ended and differential input modes. Whether this is useful is up to you ;-).

Variable autoGainAdjust
Each bit in this variable will enable automatic gain adjust for the corresponding input mode. So if bit 1 is set, the input enabled in bit 1 of variable inputSelect will have automatic gain adjust.

Variable inputGain
Each *nibble* (= group of 4 bits) will set the gain for the corresponding input mode. So *nibble* 1 will set the gain for the input mode selected with *bit* 1 in variable inputSelect.
The way the gain is set, depends on whether automatic gain adjust is disabled or enabled:

- If auto gain adjust is *disabled*, gain is *fixed* to the value defined the nibble in variable inputGain;
- If auto gain adjust is *enabled*, gain is automatically adjusted, but will never increase beyond the value defined in the nibble in variable inputGain.

Possible gain settings are:

value	Gain (FSR = Full Scale Range)
0x0	FSR = ±6.144 V, 0,003000V/digit
0x1	FSR = ±4.096 V, 0,002000V/digit
0x2	FSR = ±2.048 V, 0,001000V/digit
0x3	FSR = ±1.024 V, 0.000500V/digit

0x4 FSR = ±0.512 V, 0.000250V/digit
0x5 FSR = ±0.256 V, 0.000125V/digit
Since there are only 6 possible values, the most significant bit of the nibble is always 0.

Example
In the variable inputSelect defined at the top of the previous page, input mode 1 (AIN0 -> AIN3) is selected with bit 1 and input mode 2 (AIN1 -> AIN3) is selected with bit 2.
For both input modes auto gain adjust is enabled in variable autoGainAdjust.
Nibble 1 in variable inputGain is set to 0x1. Since auto gain adjust is enabled for this input mode, the maximum allowable gain for input mode 1 (AIN0 -> AIN3) is 0x1, which is a full scale range of +/-4.096 Volt. So the auto gain adjust can only scale between +/-6.144V and +/-4.096V.
Nibble 2 in variable inputGain is set to 0x3. Since auto gain adjust is enabled for this input mode, the maximum allowable gain for input mode 2 (AIN1 -> AIN3) is 0x3, which is a full scale range of +/-1.024 Volt. Therefore the auto gain adjust can scale between +/-6.144V and +/-1.024V.

Create an instance:

```
ADS1015_async ADS(I2C_ADS1015,
                  ADS1015_inputSelect,
                  ADS1015_autoGainAdjust,
                  ADS1015_inputGain);
```

Methods:

```
Byte result = ADS.begin();
```

This will:

- check if the ADS1015 can be found at the specified I2C address;
- start the first conversion.

A byte value is returned. A byte value of 0 means everything is OK. A value > 127 is an error condition.

```
byte result = ADS.poll();
```

The poll method will check if the ADS1015 has finished the conversion:

- the number 0 is returned if the conversion is still running;
- a number > 127 return if there was an error situation;
- if the conversion has completed, a value of 1 … 8 is returned, representing the input number for which the result is available. Note: the input number is the bit number defined in variable inputSelect **+ 1**. So if 2 is returned, this is the input enabled in bit 1 of variable inputSelect.

Note 1: If the conversion is completed and before returning the input number for which the result is available, the result is buffered and the method will start a *new* conversion for the *next* input as defined in variable inputSelect. So the ADS105 is already performing the next conversion in the background when returning from this method.
Note 2: Since the method buffers only one result, you have to fetch the result before the next result can be made available. So if you don’t get the result, the method cannot continue starting new conversions.
Note3: With automatic gain adjustment enabled and the input is rapidly rising, an overflow can occur in the current scale. The library detects this situation and will scale down to the lowest gain. The result is considered invalid, will be skipped and *not* returned.

```
float voltage = ADS.getVoltage();
```

This will return the voltage of the input number returned by the poll method. Reading the Voltage enables the poll method to make the next conversion which was running in the background, available.

```
float gain = ADS.getGain();
```

This method returns the gain, used for calculating the voltage.
Note: calling this method is optional, but if you do, call it before fetching the voltage. The reason for this is, that the method getVoltage() will release the poll function to make a new result available.

```
byte precision = ADS.getPrecision();
```

This method returns the precision of the voltage and gain values. The precision is the number of decimal numbers after the decimal point.
Note: calling this method is optional, but if you do, call it before fetching the voltage. The reason for this is, that the method getVoltage() will release the poll function to make a new result available.

Example sketch

An example sketch is included in the library.

Comparing the Adafruit and asynchronous library

Feature	Adafruit	ADS1015_async
Single ended AIN0, 1, 2, 3	√	√
Differential AIN0-1, AIN2-3	√	√
Differential AIN0-3, AIN1-3	x	√
Read a specific input	√	x
Random reading of inputs	√	x
Sequential reading of inputs	x	√
Wait until conversion result is available	**Yes**(4)	**No**
Adjustable gain	√	√
Configuration of fixed gain per selected input	x	√
Automatic gain adjust	x	√
Overflow detection	x	√(3)
Comparator functionality	√	x
5V Power supply	√	√
3.3V Power supply	(1)	(1)
Conversion time	1.983 ms	1.645 ms(2)

(1) Not tested, but it should work
(2) This is the minimum time for a conversion to become available.
(3) Only if Auto Gain Adjust is enabled for the input.
(4) But it does not check if the ADS1015 actually has finished the conversion.