How to add an ADC to Raspberry Pi: What you need to know

Raspberry Pi lacks analog input. This puts it at a disadvantage compared to microcontroller-based boards like the Arduino.

However, don't despair, you have many other options worth considering. You can **get up and running with a Raspberry Pi and an external ADC** .

Why do we need more input?

The real world is full of phenomena that, if you have the right circuit, you can easily describe using voltage. Get those voltages into digital form and you can record them, manipulate them, and use them to control other parameters and devices.

You might be looking to monitor soil moisture, greenhouse temperature, or hamster weight. You might be looking to add a volume control to your Pi, build an entire fader array, or design a joystick from scratch… The possibilities are nearly limitless.

ADC options

So which ADC is best for beginners?

The MCP3004 chip is one of the most popular and simple options from Microchip. You will have 4 or 8 10-bit channels, which can read up to 200 kSPS. On the other hand, there are Texas Instruments ADS111x devices, which read 16 bits at 860 SPS. So there's a trade-off between speed and accuracy (and of course that includes

price).

Many microcontrollers come with an inbuilt ADC. The ATMega you find on the average Arduino will provide several 10-bit channels. This is what allows the Arduino to provide analog inputs that the Raspberry Pi cannot. If you've got an Arduino involved in the setup and 10 bits is enough fidelity then this might actually be the easiest way to go.

What is Programmable Gain Amplifier?

This chip comes with some interesting features, including Programmable Gain Amplifier (PGA). It will allow you to set the desired value range according to the specification, down to a fraction of a volt. Given the number of values ??that 16 bits can represent, this allows you to detect differences of just a few microvolts.

The advantage here is that you can change the gain midway through the program. Other chips, like the MCP3004, come with an extra battery so you can provide a reference voltage.

What is multiplexing?

A multiplexer (or mux) is a switch that allows you to read multiple inputs using a single ADC. If your ADC chip has multiple input pins then there will be some internal multiplexing going on. The ADS1115's mux allows four inputs which you can select via internal registers.

Working with registers

The ADS1115 offers these options and several others. You can handle the multiplexer, adjust the gain, activate the built-in comparator, change the sample rate, and put the device into low-power sleep mode, all with the flip of a few switches. .

Where are the switches or activation buttons? They come in packages, in the form of very small bits of memory called registers. To enable a certain feature, you simply set the relevant bit to 1, not 0.

Take a look at the ADS111x datasheet, you will find these models come with 4 registers, including configuration registers, which govern the operation of the device.

Figure 36. Config Register

15	14	13	12	11	10	9	8
OS	MUX[2:0]			PGA[2:0]			MODE
R/W-1h	R/W-0h			R/W-2h			R/W-1h
7	6	5	4	3	2	1	0
DR[2:0]			COMP_MODE	COMP_POL	COMP_LAT	COMP_QUE[1:0]	

For example, bits 14 to 12 control the multiplexer. Using these three bits, you can choose from eight configurations. The number you want here is '100', which will produce the difference between the input zero and ground. On the other hand, bits 7 to 5 govern the sampling rate. If you want up to 860 samples per second, you can set these to '111'.

Once you know which option to set, you will have two bytes to send to the ADC. If you later want to set a bit here or somewhere, you can handle them individually using bitwise operators.

This is where it can get confusing. In this case, the binary does not represent a value but the values ??of the individual switches. You can express these variables as a large number, in decimal or hexadecimal form. But you should use the binary version, which is easier to read.

Connect the wire

You can plug this device straight into the breadboard. A positive voltage input will accept anywhere between 2 and 5.5v, which means the 3.3v line on the Raspberry Pi will work fine.

Wire the SDA and SCL inputs to their counterparts on the RPi and do the same things on ground and 3.3v. Take a potentiometer between the ground and voltage lines, then place the middle lead on the first input of the ADC. That's all you need to get started!

I2C handling

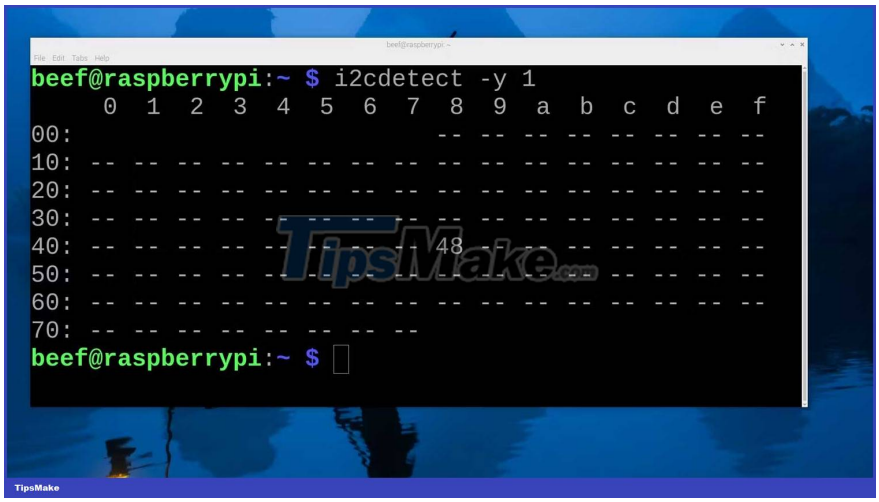
Different ADCs operate through different protocols. In the case of ADS1115, we will use I2C.

The following example will interact with ADC using Python. But before you do that, you'll need to set it up. Recent versions of Raspberry Pi OS have made this very simple. Go to Preferences > Raspberry Pi Configuration. Then, from the Interfaces tab, enable I2C.

To check everything works, open a terminal and run:

```
sudo i2cdetect -y 1
```

This command will output a grid. Assuming everything is working and you have wired it correctly, you should see a new value appear in the reticle. This is the address of the ADC. Remember that this is a hexadecimal value, so you need to add '0x' to the prefix when using it in the code below. Here it is **0x48** :



Once you have the address, you can use the SMBus library to send I2C commands. You will deal with two methods here. The first is **write_word_data()** , which accepts three arguments: the device address, the register you're writing to, and the value you want to write.

The second is **read_word_data()** , which only accepts device and register addresses. The ADC will continuously read the voltage and store the results in the conversion register. With this method you can retrieve the contents of that register.

You can beautify the result a bit and then print it. Before you return to the beginning of the loop, introduce a short delay. This will ensure you don't get overwhelmed by the data.

```
from smbus import SMBus import time addr = 0x48 bus = SMBus(1) CONFIGREG = 1 CONV
# ??c thanh ghi
b = bus.read_word_data(addr, CONVERSIONREG)
# hoán ??i hai byte
b = ((b & 0xFF)  8) | ((b >> 8) & 0xFF)
# tr? m?t n?a ph?m vi ?? ??t ti?p ??t v? 0
b -= 0x8000
# Chia k?t qu? theo ph?m vi ?? có giá tr? n?m gi?a 0 và 1
b /= TOP
# gi?i h?n ? 1
b = min(b, 1)
# phía d??i cùng là 0
b = max(b, 0)
# hai v? trí th?p phân
b = round(b, 2) print(b) time.sleep(.01)
```

You're almost done. Map the range of values ??you're getting to the value you like, then truncate the desired number of decimal places. You can adjust the print function to only print a new value when it differs from the last value.

Noise handling

This is the inherent disadvantage of using 16 bits instead of just 10 bits: a little bit of noise is more noticeable.

By connecting the adjacent input (input 1) to ground and switching the mode so that you can compare inputs one and two, you can get much more stable results. You can also swap out those noise-picking interconnect cables for small cables and add a few capacitors while you're at it. Your potentiometer value can also make a difference.

Additionally, you have software options. You can create a rolling average, or simply ignore small changes. The disadvantage is that the additional code will cause computational overhead. If you're writing conditional statements in a high-level language like Python and taking thousands of samples per second, these costs add up quickly.

It's done! Hope this article is useful to you!