Analog Devices Wiki

This version (07 Oct 2022 16:53) was ***approved*** by Mark Thoren [https://ez.analog.com/members/mthoren_adi].
The Previously approved version (/university/labs/software/precision_adc_toolbox?rev=1609708917) (03 Jan 2021 21:21) is available.

Precision ADC Tutorial

Introduction

The goal of this tutorial is to equip the reader with a collection of hardware and software tools for developing precision converter applications. That's a pretty broad statement, but then again, so is the application space for such converters. They are used for all manner of measurement applications - temperature measurement, strain gauge measurement, general-purpose industrial data acquisition. They are occasionally mis-used: - a converter with a 19.2ksps sample rate sounds like it should be suitable for digitizing a 9kHz analog signal… unless the ADC's internal digital filter gets in the way.
This tutorial will explore some representative applications, including some subtleties of such measurements - low-level thermal effects, digital filter response (including reverse engineering some of an ADC's internal filters), and understanding an ADC's noise performance.
Throughout the exercises we'll be writing simple Python code to capture and analyze data, using the industry standard Industrial I/O (IIO) framework to interact with the ADC, and the popular NumPy and Matplotlib Python libraries. Thus this exercise also serves as a mini-tutorial on Python.

Materials

- Raspberry Pi 4; 2G, 4G, or 8G version. (3B, 3B Plus will work, but you will want the 4 🙂)
- 5V USB (Universal Serial Bus)-C wall adapter for Raspberry Pi (micro USB (Universal Serial Bus) for model 3)
- EVAL-AD7124-8-PMDZ [https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad7124-8-pmdz.html]
- Electrical connection hardware (choose one):
    - 12x 15cm socket-to-socket jumpers such as these from Schmartboard [https://schmartboard.com/wire-jumpers/female-jumpers/5-inch/]
    - DesignSpark HAT to Pmod Adapter [https://reference.digilentinc.com/reference/add-ons/pmod-hat/start]
- 16GB (or larger) Class 10 (or faster) micro-SD (Secure Digital) card
- User interface setup (choose one):
    - HDMI monitor, keyboard, mouse plugged directly into Raspberry Pi
    - Host Windows/Linux/Mac computer on same network as Raspberry Pi
- ADALM2000 [https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adalm2000.html] (Not required for all experiments.)
- 10kΩ resistors (2)
- 10Ω resistor
- 1kΩ resistor
- Breadboard or prototyping board, hookup wire
- Clone or download zip of the Python code for this tutorial

Resources:

- LTspice files: Precision Converter Tutorial Python Code [https://analogdevicesinc.github.io/DownGit/#/home?url=https://github.com/analogdevicesinc/education_tools/tree/master/m2k/python/precision_adc_tutorial]

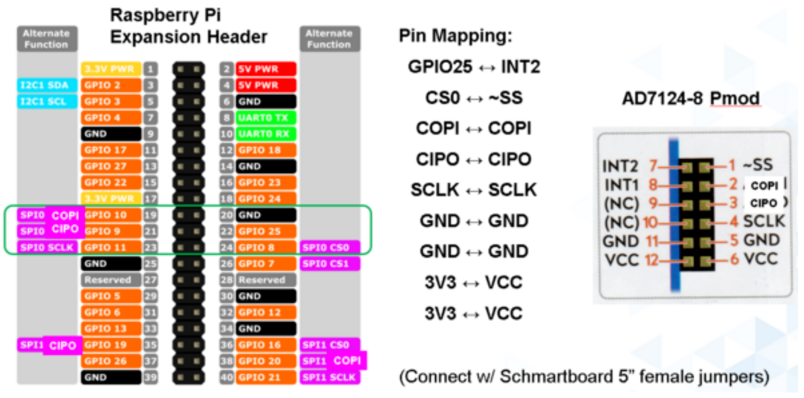
Hardware Setup: Linux / Raspberry Pi

The Raspberry Pi-based hardware and Linux setup mirrors that of the ADXL345 used in the Converter Connectivity Toolbox and Tutorial (/university/labs/software/iio_intro_toolbox), including bringing up the pyadi-iio example. Follow the instructions for downloading and installing ADI (Analog Devices, Inc.) Kuiper Linux, and editing config.txt. The only difference is the interrupt connection and device tree overlay to be added to config.txt. For this exercise, add the following line to config.txt:

```
dtoverlay=rpi-ad7124-8-all-diff-cs0-int25
```

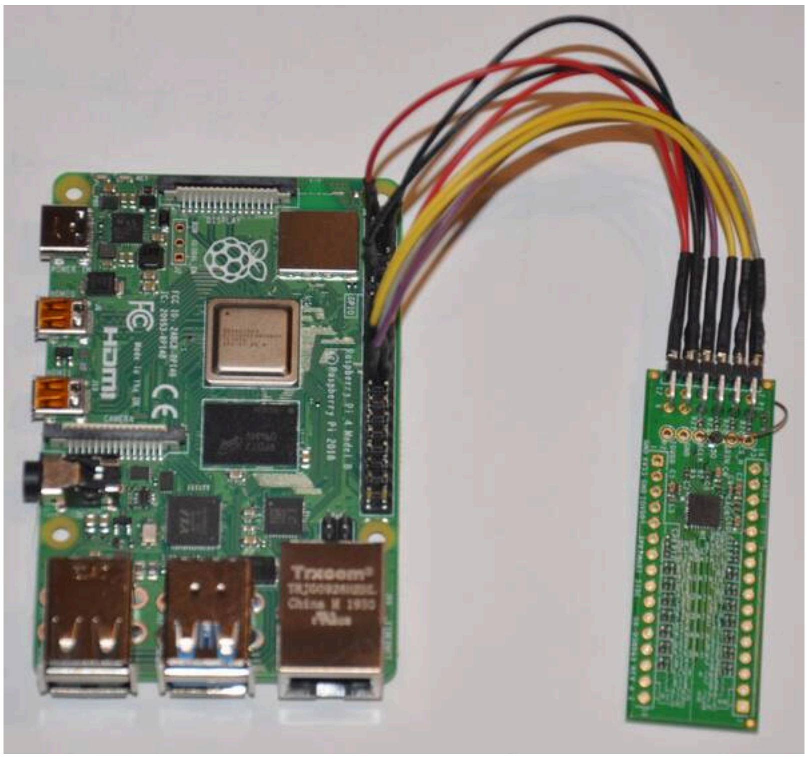
And as the filename would indicate, connect the SDO/CIPO line to Raspberry Pi GPIO25. The AD7124-8 does not have a separate interrupt pin, rather, a falling edge on SDO indicates that a conversion has finished and data is ready. The driver uses this event as an interrupt, so the SDO pin must also be connected to a GPIO (General Purpose Input/Output) pin that is configured as the interrupt source in the device tree.

The EVAL-AD7124-8-PMDZ has the option to connect the SDO line to Pmod pin 7, 8, 9, or 10, with the default being pin 7. If you are using discrete wires, simply connect Pmod pin 7 to Raspberry Pi GPIO25, following the pin mapping in Figure 1 and corresponding photo in Figure 2.



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_pmdz_connections.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

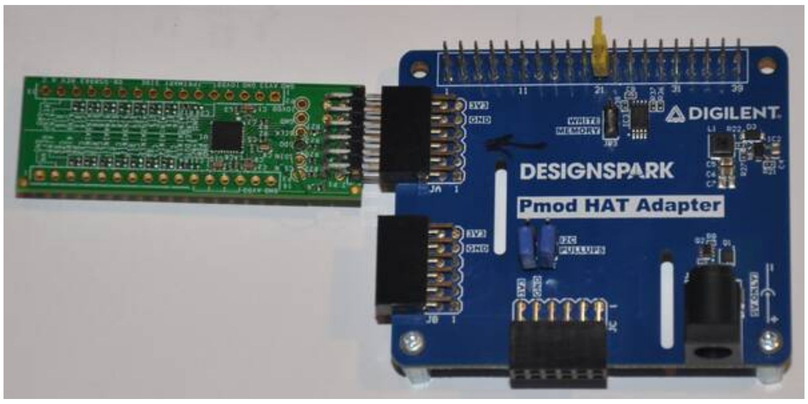
Figure 1. Raspberry Pi to Pmod Connections



(/_detail/university/labs/software/precision_adc_toolbox/rpi_connections_discrete.jpg?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 2. Connections with Discrete Wires

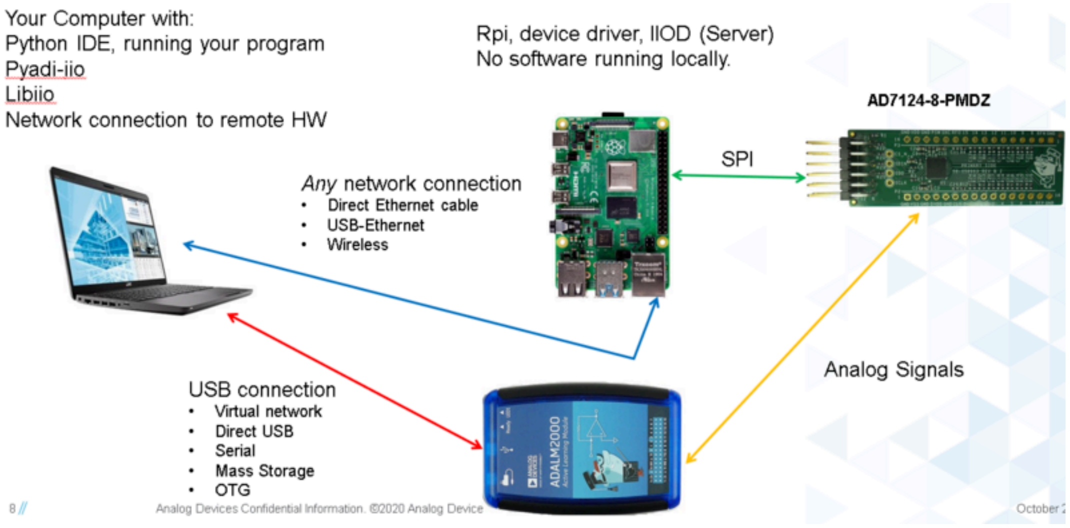
However, if you are using the DesignSpark adapter, the easiest way to make the connection is to de-solder P8 on the reverse side of the Pmod, and place a 100-mil jumper on the top-side connector, between pins 21 and 22 (GPIO9/MISO and GPIO25) as shown in Figure 3. Note that the Pmod is installed in adapter receptacle JA, which uses CS0. Alternative configurations are possible with this adapter, but will require modifications to the device tree overlay (which will be covered later.)



(/_detail/university/labs/software/precision_adc_toolbox/rpi_connections_adapter.jpg?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 3. Connections with DesignSpark adapter

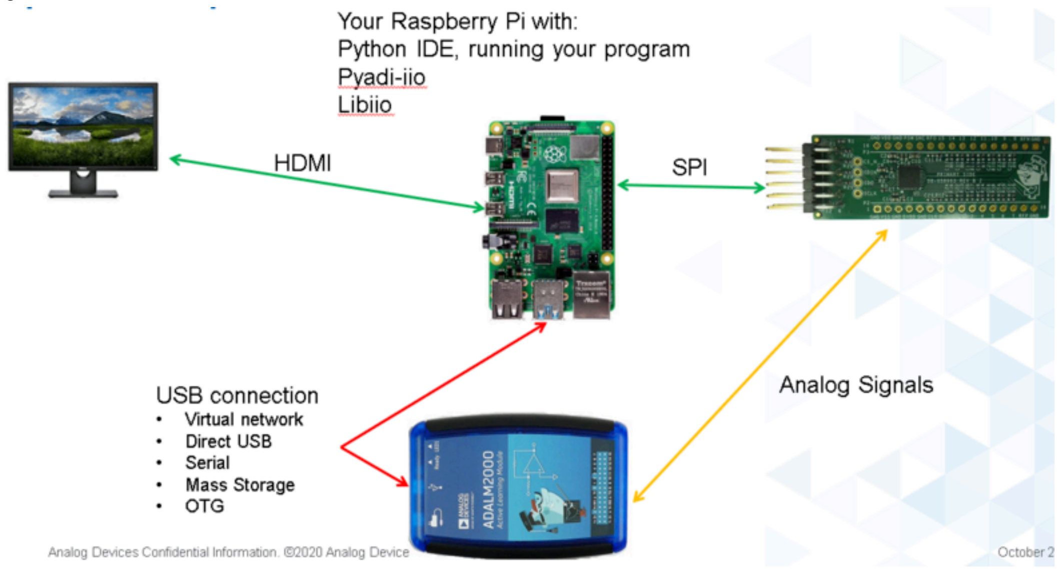
Figure 4 shows a “full featured” development setup, typical of an application in which the Raspberry Pi is acting as a “bridge” between a more powerful host computer and the ADC. The ADALM2000 shown could be replaced by more elaborate test instruments such as precision voltage calibration sources or benchtop multimeters, controlled by the host via GPIB, Ethernet, or other means.



(/_detail/university/labs/software/precision_adc_toolbox/full_setup_overview.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 4. "Full-Featured" Test Setup

But recall that the Raspberry Pi is a fully functional computer all by itself, so it is completely valid to get rid of the host computer entirely, and run all software on the Pi itself (Amazing!!). Refer to Figure 5.



(/_detail/university/labs/software/precision_adc_toolbox/pi_only_setup_overview.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 5. Test Setup Using Only Raspberry Pi

Hardware Setup: no-OS / ADICUP3029

A Tinyiiod implementation for the AD7124-8-PMDZ + ADICUP3029 is in the works. Describe here, and make sure to add how to connect to the serial backend via libiio/pyadi-iio
Photo of AD7124-8-PMDZ plugged into ADICUP3029

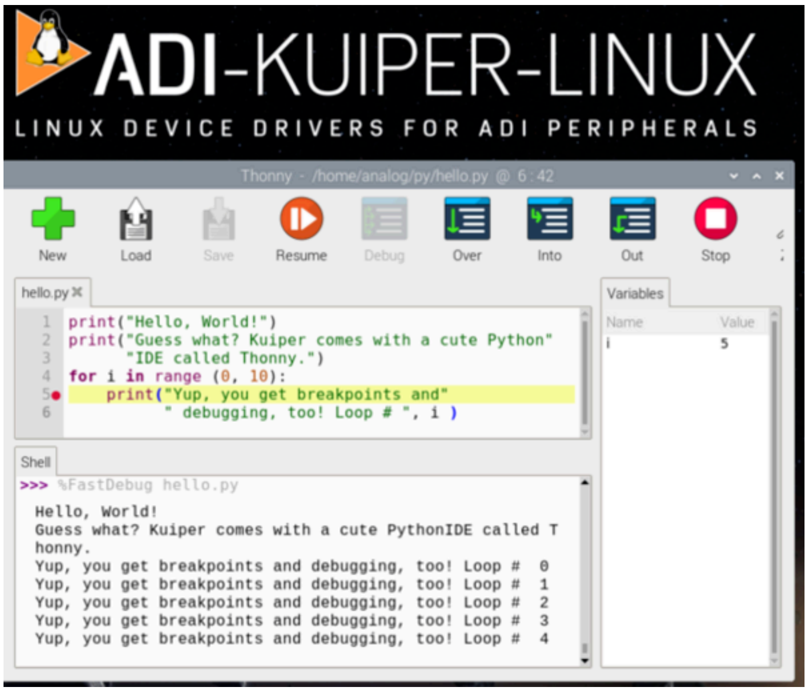
Toolbox Item: Python Environment and Libraries

First, if you've never touched Python before (or need a refresher), a great resource is learnpython.org [https://www.learnpython.org/]. All exercises run directly in the browser so there's nothing to install, and going through the "Learn the Basics" examples will give enought of a background for this tutorial. (Or, just dive right in and run this tutorial's code, and learn the basics later.)

But in order to communicate with hardware, you'll need an installed Python environment. There are numerous Python distributions and integrated development environments (IDEs) available. A few popular ones are:

Anaconda [https://www.anaconda.com/]
PyCharm [https://www.jetbrains.com/pycharm/]
Jupyter Notebook [https://jupyter.org/]
Thonny ("Python IDE for Beginners") [https://thonny.org/]

Each of these has its merits, but there is a lot to be said for choosing one that is already being used by your company/school/co-workers (in case you need to ask for help). And if the idea of choosing and installing a Python environment sounds daunting, Thonny is already preinstalled in ADI (Analog Devices, Inc.) Kuiper Linux, as are all required libraries for this tutorial. So if you are logged into your Raspberry Pi, run Thonny from the start menu (under "Programming"), and enter the snippet of code shown in Figure 6. Try setting a breakpoint in the loop, and step through an iteration at a time, noting the value of the variable “i”.



```
print("Hello, World!")
print("Guess what? Kuiper comes with a cute Python"
      "IDE called Thonny.")
for i in range (0, 10):
    print("Yup, you get breakpoints and"
          " debugging, too! Loop # ", i )
```

```
>>> %FastDebug hello.py
 Hello, World!
 Guess what? Kuiper comes with a cute PythonIDE called T
 honny.
 Yup, you get breakpoints and debugging, too! Loop #  0
 Yup, you get breakpoints and debugging, too! Loop #  1
 Yup, you get breakpoints and debugging, too! Loop #  2
 Yup, you get breakpoints and debugging, too! Loop #  3
 Yup, you get breakpoints and debugging, too! Loop #  4
```

(/_detail/university/labs/software/precision_adc_toolbox/thonny_example.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 6. Thonny Python IDE (Integrated Drive Electronics (hard drives!))

A few other libraries are required for interacting with the EVAL-AD7124-8-PMDZ and ADALM2000:

- libiio - Library for interfacing with Linux IIO devices
- pyadi-iio - Python Interfaces For IIO Drivers
- libm2k - Library for interfacing with the ADALM2000, including Python bindings.

All of these are preinstalled on ADI (Analog Devices, Inc.) Kuiper Linux, so if you're runing this exercise on the Pi itself, there's nothing else to install.

Brief installation instructions for these on remote host, point to wiki pages: libiio
pyadi-iio
libm2k

Toolbox Items: NumPy and Matplotlib

NumPy

From NumPy.org: [https://numpy.org/]
“NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.”
These exercises will only use a few basic NumPy functions, such as:

```
np.zeros     #(Make me an array of zeros)
np.ones      #(Make me an array of ones)
np.fft       #(Take FFT of my data)
np.abs       #(Magnitude of complex data, use w/ fft)
np.convolve  #(Slide a filter across my data)
np.random    #(Make me some random data)
np.std       #(Calculate the standard deviation of my data)
```

But do explore NumPy.org, many of the functions include example scripts that can be run as-is. And as usual, NumPy is already installed on ADI (Analog Devices, Inc.) Kuiper Linux. NumPy is also bundled with many of the popular Python distributions. To see if NumPy is installed, enter the following from the Python console (this example is run from a terminal on the Raspberry Pi, including running Python 3):

```
analog@analog:~ $ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.version.version
'1.15.1'
>>>
```

If the import fails, or if the version is prior to 1.19.2, NumPy can be installed or upgraded by running:

```
(sudo) apt install python3-numpy python3-scipy
```

on Linux systems (including Raspberry Pi):

```
pip install numpy
```

or if you're using Anaconda:

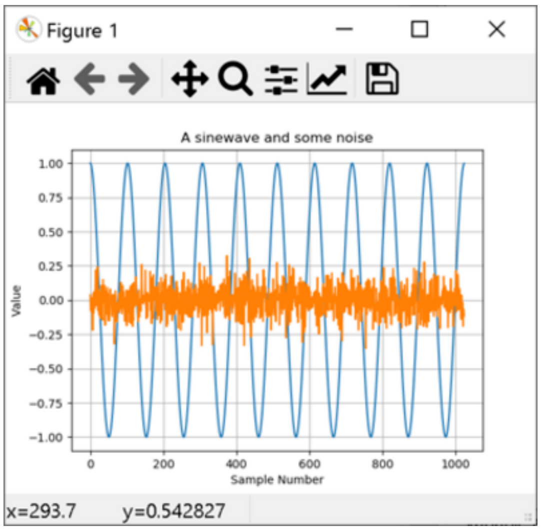
```
conda install numpy
```

Matplotlib

From matplotlib.org: [https://matplotlib.org/]
“Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.” Matplotlib is useful for instant gratification from within your Python script – but you can also send the same data to some other plotting routine, or save data for later analysis in another program.

Warmup: Let's make some waves! Let's make some noise!

With the Python envrironment all set up, let's start running some examples. At this point you should have either cloned or downloaded the code for this exercise from the Github link in the materials list. Open up the **make_noise_and_waves.py** script in Thonny (or your favorite Python environment) and have a look around. Run the script, and you should see a Matplotlib plot show up similar to Figure 7. Well, the cosine wave should be EXACTLY as shown, or at least indistinguishable to the human eye. However, the noise will change slightly each time the script is run, as the computer uses various sources of (almost) random numbers as a starting point for generating the array.



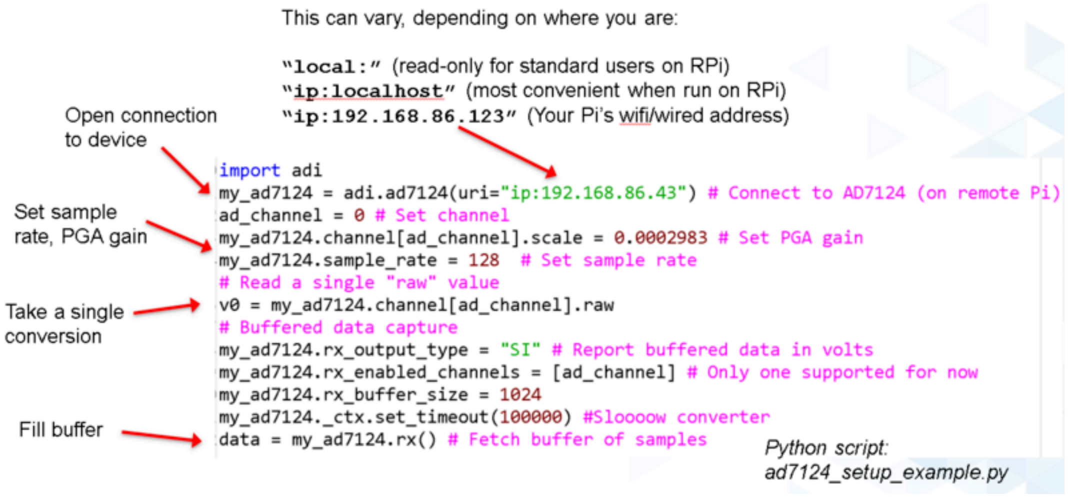
(/_detail/university/labs/software/precision_adc_toolbox/noise_and_waves_plot.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 7. Noise and Waves

Experiment with changing the frequency and amplitude of the cosine wave, change to sine (or if you're adventurous, tangent, but note that you'll need to manually set the vertical axis of the plot to avoid shooting off to (almost) infinity).
Experiment with the amplitude and offset of the noise values as well.

Toolbox Item: Pyadi-iio

Pyadi-iio is a python abstraction module for ADI (Analog Devices, Inc.) hardware with IIO drivers to make them easier to use. Complete documentation for Pyadi-iio can be found here [https://analogdevicesinc.github.io/pyadi-iio/]. Pyadi-iio is already installed on ADI (Analog Devices, Inc.) Kuiper Linux, or you should have installed it on your host machine as noted earlier. Figure 8 shows a snippet of code for the AD7124 interface specifically. This snippet is included in the examples, as **ad7124_setup_example.py**. Open this script in your Python IDE (Integrated Drive Electronics (hard drives!)) and run it. If the script runs without error, congratulations, you're talking to your AD7124!


Open connection to device
Set sample rate, PGA gain
Take a single conversion
Fill buffer
Python script: ad7124_setup_example.py

This can vary, depending on where you are:

"local:" (read-only for standard users on RPi)
"ip:localhost" (most convenient when run on RPi)
"ip:192.168.86.123" (Your Pi's wifi/wired address)

```
import adi
my_ad7124 = adi.ad7124(uri="ip:192.168.86.43") # Connect to AD7124 (on remote Pi)
ad_channel = 0 # Set channel
my_ad7124.channel[ad_channel].scale = 0.0002983 # Set PGA gain
my_ad7124.sample_rate = 128  # Set sample rate
# Read a single "raw" value
v0 = my_ad7124.channel[ad_channel].raw
# Buffered data capture
my_ad7124.rx_output_type = "SI" # Report buffered data in volts
my_ad7124.rx_enabled_channels = [ad_channel] # Only one supported for now
my_ad7124.rx_buffer_size = 1024
my_ad7124._ctx.set_timeout(100000) #Slooooow converter
data = my_ad7124.rx() # Fetch buffer of samples
```

(/_detail/university/labs/software/precision_adc_toolbox/ad7124_pyadi_iio.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 8. AD7124 Pyadi-iio interface details

Note that the uri (Uniform Resource Identifier) depends on how you are connecting to the device:

- local: when running the script on the same machine to which the AD7124 is connected, and you have sufficient privileges
- ip:localhost when running locally, but communicating through iiod via the local loopback network interface, avoiding issues with privileges
- ip:www.xxx.yyy.zzz [http://www.xxx.yyy.zzz] when connecting from a remote machine
- (in the near future) serial:[port] when connecting over the serial backend, as with tinyiiod running on an embedded target.

The rest of the examples in this exercise are pretty self-explanatory, but it's not a bad idea to peruse the pyadi-iio source [https://github.com/analogdevicesinc/pyadi-iio], in particular, the adi directory (which contains the interfaces themselves) and the examples directory.

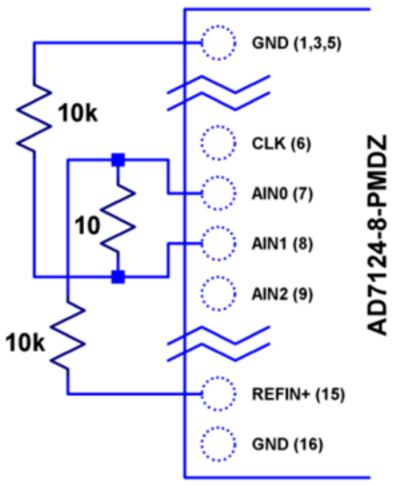
But the various functions and methods can also be listed out from the Python console, as noted below.

Tip:use the “type” and “dir” functions to explore what functions and methods exist for a particular object. Below are two examples, starting with a plain old Python list, and then a pyadi-iio instance of an AD7124-8:

```
analog@analog:~ $ python3
>>> a=[1,2,3]
>>> print(a)
[1, 2, 3]
>>> type(a)
<class 'list'>
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
...
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
>>>
>>> myadc=adi.ad7124("ip:localhost") # Connect to AD7124 (hardware required)
>>> type(myadc)
<class 'adi.ad7124.ad7124'>
>>> dir(myadc)
['__annotations__', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
...
'_rx_unbuffered_data', '_rxadc', '_set_iio_attr', '_set_iio_attr_float', '_set_iio_attr_int',
'_set_iio_debug_attr_str', '_set_iio_dev_attr_str', '_uri_auto', 'channel', 'rx', 'rx_buffer_size',
'rx_enabled_channels', 'rx_output_type', 'sample_rate', 'scale_available', 'to_volts', 'uri']
>>>
>>> len(myadc.channel) # See how many channels there are as configured
8
>>> myadc.channel[0].raw # Grab a raw reading
8535
>>>exit() # Quit Python
```

Experiment: Measuring the AD7124-8 Input Noise

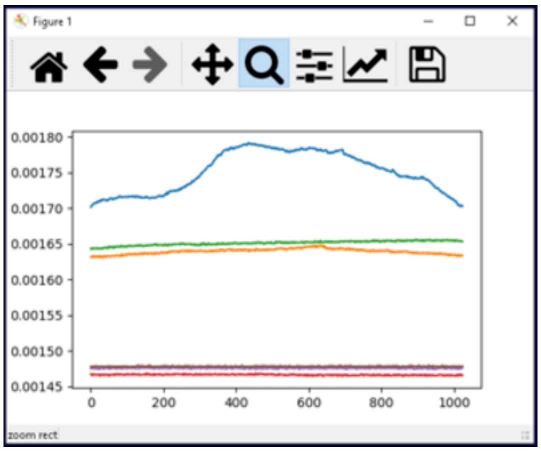
Now that we've established communication to the AD7124-8, let's run an extremely simple, yet extremely useful test - measuring input noise. Simply shorting the input to an ADC and looking at the resulting distribution of ADC codes is a valuable (arguably essential) step in validating a signal chain design. One subtlety about the configuration as set by the rpi-ad7124-8-all-diff-cs0-int25 overlay is that the input range is unipolar, so only positive values are valid. (It is still *differential*, meaning, the measurement is taken BETWEEN adjacent inputs.) This means that a converter with perfect offset will produce a “half historgram” output, with half of the values equal to zero (because that's the lowest valid output value), and half of the values slightly above zero. The solution is to apply a very small input voltage that overcomes the offset, but does not add significant noise. Build the circuit shown in Figure 9, which will impose a 1.25mV signal across the input (far larger than the 15µV uncalibrated offset of the AD7124-8.)



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_noise_circuit.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 9. AD7124 input offset circuit

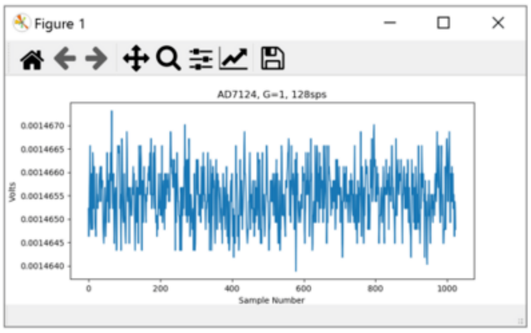
Open the **ad7124_basic_capture.py** script in your Python IDE (Integrated Drive Electronics (hard drives!)) and run it. You should see an output plot similar to Figure 10.



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_warmup.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 10. AD7124 noise test during warmup

If you run the script a couple of times right after turning on the power, you may see some drift or “wandering”. This can be due to a number of factors - the internal reference warming up, the external resistors warming up (and hence drifting), or even parasitc thermmocouples, where slightly dissimilar metals will produce a voltage in the presence of thermal gradients. The lower traces in Figure 10 are after wrapping the AD7124 and resistor divider in antistatic bubble wrap, and waiting half an hour. Finally, Figure 11 shows a single trace after warmup.



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_time_noise.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 11. AD7124 noise, 120sps mode

The captured data is still available in the Python console, so we can do some quick analysis:

```
>>> %Run ad7124_simple_capture.py
0.000149011
>>> np.std(data)
5.658575585362495e-07
>>>
```

Which tells us that the noise level is about 565nVRMS, about right for 128 samples per second.

Double-plus verify exactly which filter mode we're in, based on https://github.com/analogdevicesinc/linux/blob/master/drivers/iio/adc/ad7124.c#L266 [https://github.com/analogdevicesinc/linux/blob/master/drivers/iio/adc/ad7124.c#L266]

At this point, we can be confident that at least things have a chance of going right, and we can start connecting sensors or additional signal conditioning circuitry.

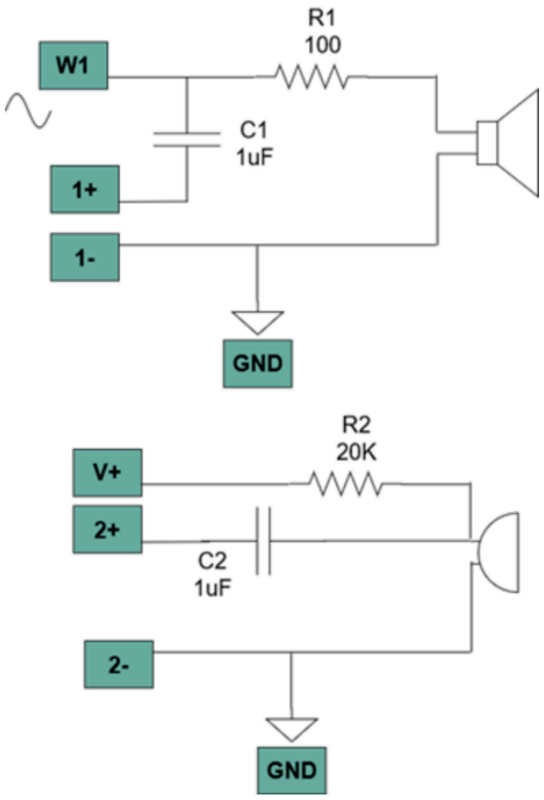
Toolbox Item: Libm2k

The signal source for the input noise mesaurement was easy - it was simply the noise inherent in the ADC itself. But it's useful to be able to send idealized test signals to the ADC to verify that it will work in the intended application, whether a low-speed measurement like temperature, or something higher speed, like a dynamic weigh scale. There are various benchtop instruments that will do that job, but the ADALM2000 [https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adalm2000.html] multifunctional USB (Universal Serial Bus) test instrument (or, “m2k” for short) is very handy for getting started, even if it will eventually replaced with something more elaborate.

Experiment: libm2k Voice Reverser

Before we hook up the AD7124, let's try something fun that demonstrates capturing a waveform with the m2k, processing with NumPy functions, then playing back. The example is an audio flipper that records a few seconds of audio then plays it back backwards. (Yes this is a bit indulgent and you can skip it if you want, but at least atke a look at the code.)

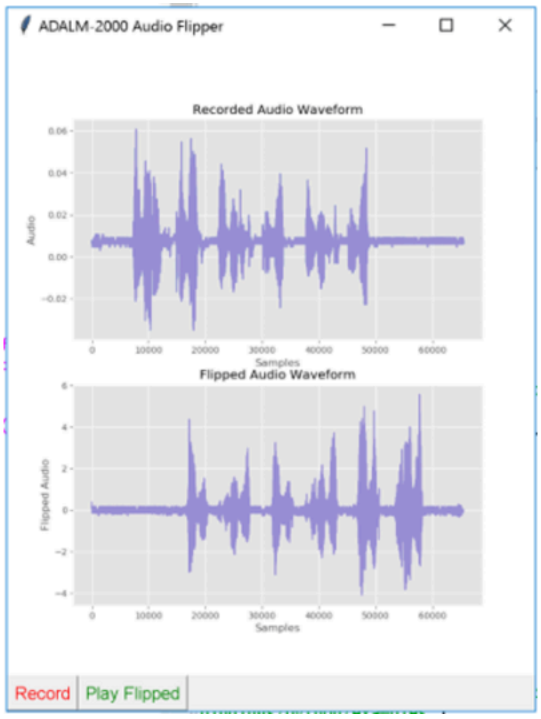
Build up the circuit shown in Figure 12.



(/_detail/university/labs/software/precision_adc_toolbox/audio_flip_circuit.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 12. Microphone / Speaker circuit for Audio Flipper

Open **audio_flip_with_gui.py** in your Python IDE (Integrated Drive Electronics (hard drives!)) and run it. A little GUI (Graphical User Interface) will pop up with a record and playback button. Have fun!



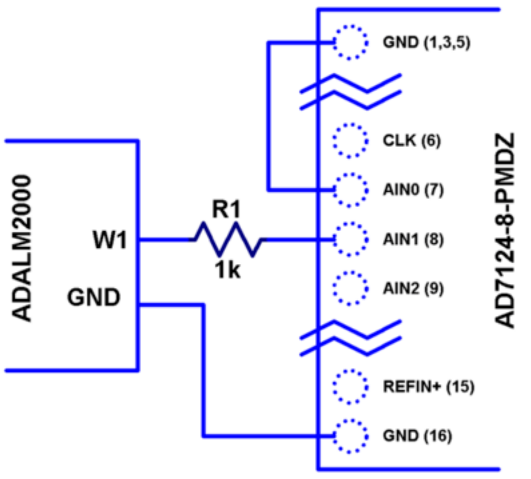
(/_detail/university/labs/software/precision_adc_toolbox/audio_flip_gui.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 13. Audio Flip GUI (Graphical User Interface)

Experiment: Measuring the response of the AD7124 filter

The AD7124-8 is a sigma-delta ADC, in which a modulator produces a high sample rate, but noisy (low resolution), representation of the analog input. This noisy data is then filtered by an internal digital filter, producing a lower rate, lower noise output. The type of filter varies widely depending on the intended end application - an audio sigma-delta ADC will have a filter that is flat out to 20kHz, with an output data rate of at least 44ksps. The AD7124-8 is general-purpose, targetted at precision applications. As such, the digital filter response and output data rate are highly configurable. While the filter response is well-defined in the datasheet, there are occasions when one may want to measure the impact of the filter on a given signal. This experiment measures the filter response by applying sinewaves to the ADC input and analyzing the output. This method can be easily adapted to measuring other waveforms - wavelets, simulated physical events, etc.
Connect the ADALM2000 to the EVAL-AD7124-8-PMDZ as shown in Figure 14. The 1k resistor is to protect the AD7124-8 in case something goes wrong, as the m2k output range is

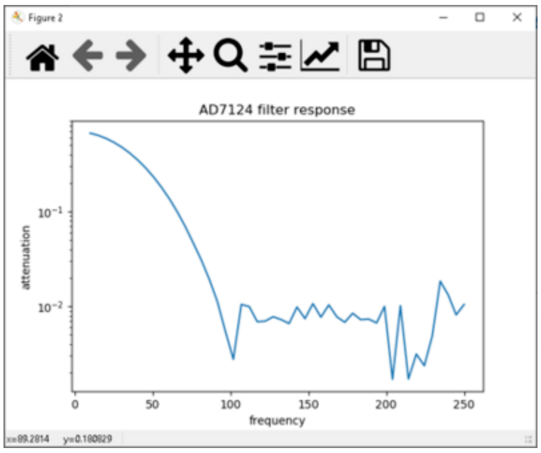
-5V to +5V, beyond the -0.3V to 3.6V absolute maximum limits of the AD7124-8. DO NOT OMIT THIS RESISTOR.



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_m2k_circuit.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 14. AD7124 - m2k Connections for Filter Response Measurement

Load **trace_ad7124_filter_with_m2k.py** into your Python IDE (Integrated Drive Electronics (hard drives!)) and run it. The script will set the m2k's waveform generator to generate a sinewave at 10Hz, capture 1024 data points, calculate the RMS value, then append the result to a list. It will then step through frequencies up to 250Hz, then plot the result as shown in Figure 15.



(/_detail/university/labs/software/precision_adc_toolbox/ad7124_filter_resp_measured.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 15. AD7124 Measured Filter Response, 128sps

Retake data, rescale vertical axis in dB (decibel)

So while it's difficult to measure high attenuation values without quite a bit more care, the response of the first couple of major “lobes” is apparent. At this point, you're all set up to send your own waveforms to the AD7124 and see how it responds, just replace the sinewave data that is pushed to the m2k with your own data.

Bonus Experiment: Reverse Engineering one of the AD7124's filters

The ability to measure an ADC's filter response is certainly a practical tool to have at your disposal. However, in order to fully simulate applications, a model of the filter is needed. This isn't explicitly provided for the AD7124-8, so we'll take this opportunity to try to reverse engineer it from the information provided in the datasheet.

warning What follows is only a model of the AD7124-8 filters, it is not a bit-accurate representation. Refer to the AD7124-8 datasheet for all guaranteed parameters.

Figures 16 and 17 show the AD7124-8's 10Hz and 50Hz notch filters. Various combinations of Higher order SINC3 and SINC4 filters are also available.

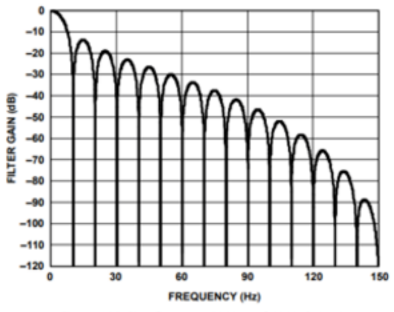
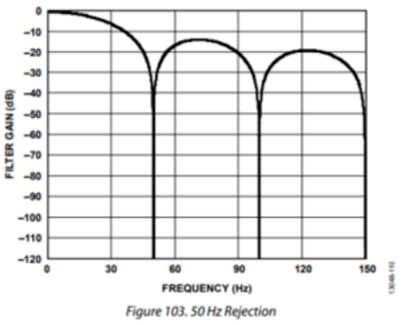


Figure 105. Simultaneous 50 Hz and 60 Hz Rejection

(/_detail/university/labs/software/precision_adc_toolbox/ad7124_filter_10.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 16. AD7124-8 10Hz notch filter

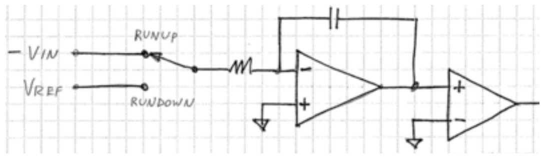


(/_detail/university/labs/software/precision_adc_toolbox/ad7124_filter_50.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 17. AD7124 50Hz notch filter

An Analog Averaging Circuit

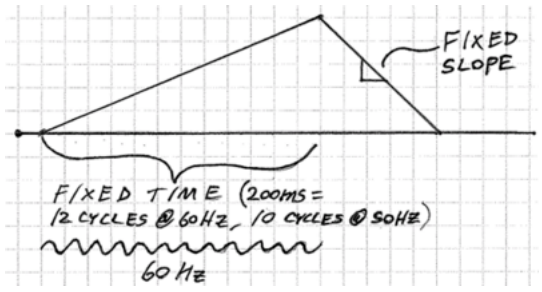
Before we dig any deeper into the AD7124 digital filters, let's take a step back and look at a circuit that's still ubiquitous (in various forms) in high-perofmance benchtop meters: the dual-slope ADC, shown in Figure 18. Without going too deep into the details, a dual slope converter observes and averages an input voltage for a fixed time period, where the input voltage controls the slope of an integrator's output. The integrator is then switched to a known reference voltage of opposite sign, causing the integrator to ramp back to its starting voltage.
If you **ARE** interested in digging deeper into the operation of a dual-slope converter, see the dual-slope section of the Analog to Digital Conversion Active Learning Activity (/university/courses/electronics/electronics-lab-adc)



(/_detail/university/labs/software/precision_adc_toolbox/dual_slope_schematic.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 18. Dual-slope converter schematic

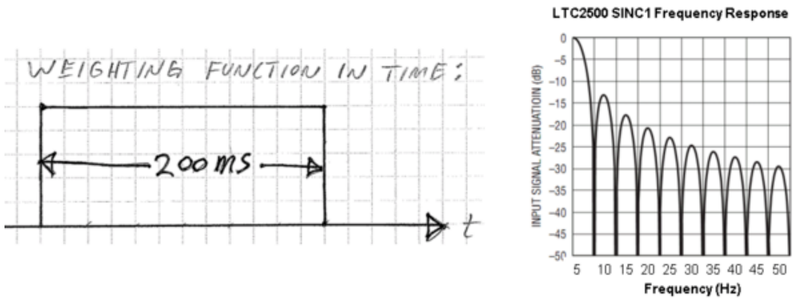
The ratio of the de-integration ("rundown") time and integration ("runup") time is then equal to the ratio of the input voltage to the reference voltage, which can then be scaled and displayed.



(/_detail/university/labs/software/precision_adc_toolbox/dual_slope_operation.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 19. Dual Slope Operation

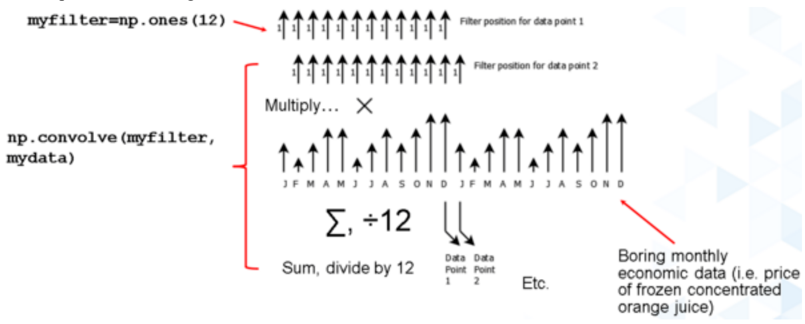
This results in a SINC1 lowpass filter response as shown in Figure 20.



(/_detail/university/labs/software/precision_adc_toolbox/sinc1_time_and_frequency.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 20. SINC1 Filter, Impulse Response and Frequency Response

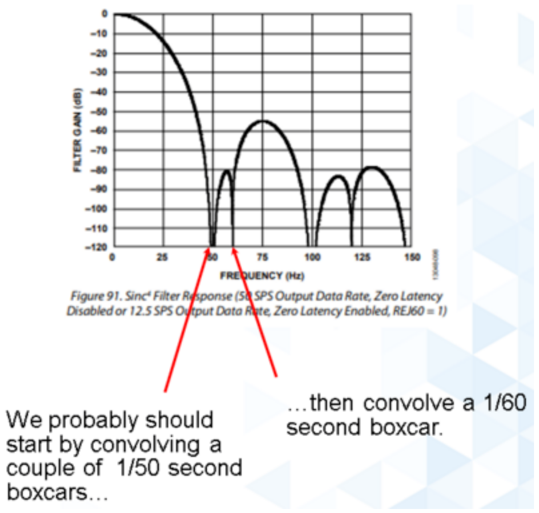
The dual-slope circuit ***continuously*** samples the input signal, so it is an "analog SINC1 filter". But a similar idea in the discrete-time domain is that of a "running average" often heard applied to economic data. Figure 21 shows the price of Frozen Concentrated Orange Juice (FCOJ) for a few months. For each new data point, average it with the previous 11 month's values, and produce a new "filtered" data point. Notice that any yearly, bi-yearly, quarterly, etc. fluctuations will be "nulled out" and will not appear in the output data set. This is a digital (or discrete-time) SINC1 filter, similar in concept to those in a sigma-delta ADC.


myfilter=np.ones(12)
Filter position for data point 1
Filter position for data point 2
np.convolve(myfilter, mydata)
Multiply... ×
J F M A M J J A S O N D J F M A M J J A S O N D
∑, ÷12
Sum, divide by 12
Data Point 1
Data Point 2
Etc.
Boring monthly economic data (i.e. price of frozen concentrated orange juice)

(/_detail/university/labs/software/precision_adc_toolbox/12_month_running_average.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 21. FCOJ price and filtering operations

Next, let's see if we can reverse-engineer one of the AD7124's internal filters. And to keep it interesting we'll choose one with a strange frequency response, like the simultaneous 50Hz/60Hz rejection filter shown in Figure 22.



(/_detail/university/labs/software/precision_adc_toolbox/simult_50_60_reverse_eng.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 22. AD7124-8 50/60Hz rejection filter

Higher order SINC filters can be generated by convolving SINC1 filters. For example, convolving two SINC1 filters (with a rectangular impulse response in time) will result in a SINC2 response, with a triangular impulse response.
Load **ad7124_filters.py** into your Python IDE (Integrated Drive Electronics (hard drives!)), and before running it take a look through the code. In particular, the following snipped derives a SINC3 filter with a null at 50Hz:

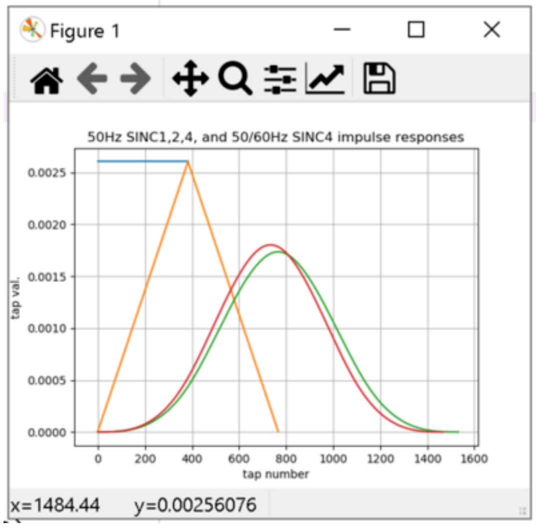
```
f0 = 19200
# Calculate SINC1 oversample ratios for 50, 60Hz
osr50 = int(f0/50) # 384
osr60 = int(f0/60) # 320

# Create "boxcar" SINC1 filters
sinc1_50 = np.ones(osr50)
sinc1_60 = np.ones(osr60)

# Calculate higher order filters
sinc2_50 = np.convolve(sinc1_50, sinc1_50)
sinc3_50 = np.convolve(sinc2_50, sinc1_50)
sinc4_50 = np.convolve(sinc2_50, sinc2_50)

# Here's the filter from datasheet Figure 91,
# SINC4-ish filter with one three zeros at 50Hz, one at 60Hz.
filt_50_60_rej = np.convolve(sinc3_50, sinc1_60)
```

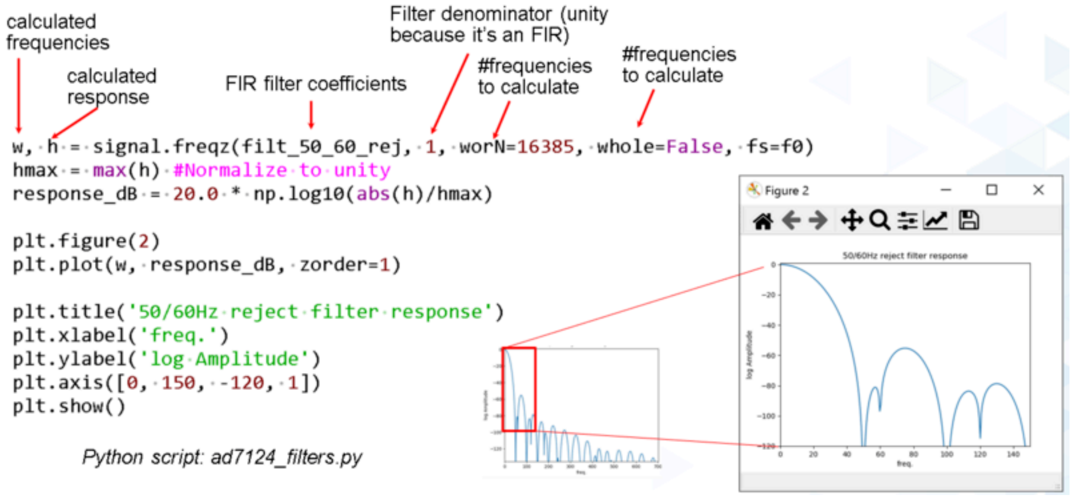
Run the script, and observe the impulse (time domain) shapes of the filters, shown in Figure 23.



(/_detail/university/labs/software/precision_adc_toolbox/rev_eng_filters_all.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 23. Generated Filter Impulse Responses

And finally, let's calcualte the frequency response using NumPy's freqz function, shown in Figure 24.


calculated frequencies
calculated response
FIR filter coefficients
Filter denominator (unity because it's an FIR)
#frequencies to calculate
#frequencies to calculate
Python script: ad7124_filters.py
Figure 2
50/60Hz reject filter response
log Amplitude
freq.

```
w, h = signal.freqz(filt_50_60_rej, 1, worN=16385, whole=False, fs=f0)
hmax = max(h) #Normalize to unity
response_dB = 20.0 * np.log10(abs(h)/hmax)

plt.figure(2)
plt.plot(w, response_dB, zorder=1)

plt.title('50/60Hz reject filter response')
plt.xlabel('freq.')
plt.ylabel('log Amplitude')
plt.axis([0, 150, -120, 1])
plt.show()
```

(/_detail/university/labs/software/precision_adc_toolbox/freqz_annotated.png?id=university%3Alabs%3Asoftware%3Aprecision_adc_toolbox)

Figure 24. Calculated Frequency Response Using Freqz

Conclusion

It's important to keep in mind that this filter is only a model, not bit-accurate to what the AD7124-8 does internally. But it gives you a pretty good idea of what to expect, and can be used as a reality check on simulated data for a particular application, like a chicken moving around on a weigh scale, or pulses from a blood pressure cuff.