ADCplate Users Guide

Table of Contents

Overview

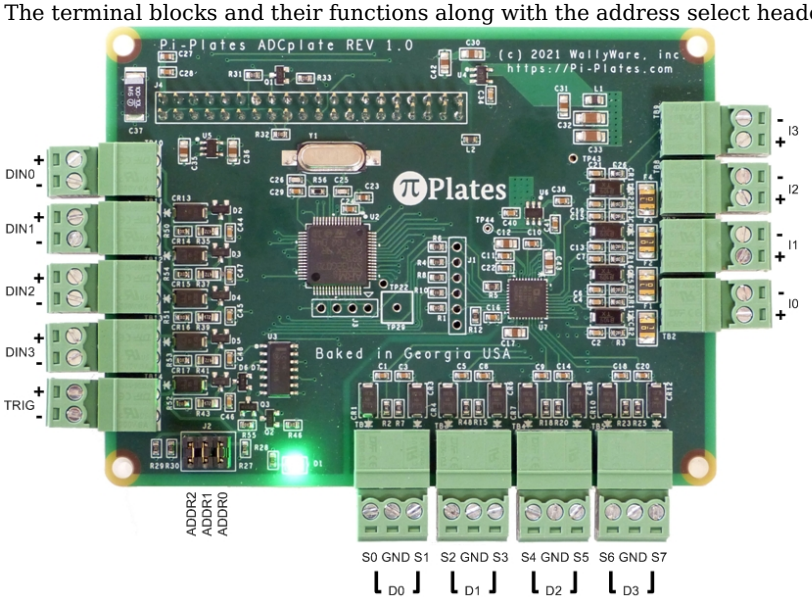
This page describes all of the features and functions of the Pi-Plates ADCplate. This Pi-Plate operates in two fundamental modes: Easy add Advanced.

All of the examples below were written for Python 3 and require that the ADCplate module is imported with the following statement:

import piplates.ADCplate as ADC

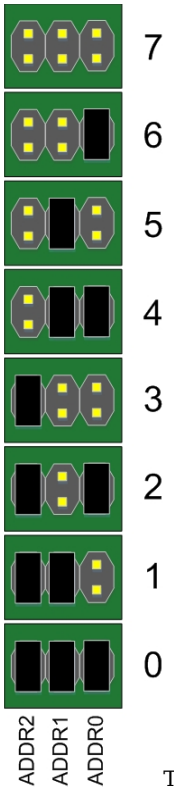
Board Layout

The terminal blocks and their functions along with the address select header are shown below:



Address Selection Header

Up to eight ADCplates can be used in a single stack of Pi-Plates. To do this, each board has to be set to a unique address. When shipped, the ADCplate is set to address zero. The address is set by positioning jumpers on the small, six pin header on the lower left of board as shown in the image above. Use the diagram below to set the address:



The address headers can be changed at any time and do not require that the ADCplate be powered off.

Basic Operation

Modes

At power up or after a reset, the ADCplate operates in “Easy” mode. In this mode, data from each analog input is continuously acquired and available to read at any time. This makes it easy to bring up your code and start collecting data quickly. There are advantages and disadvantages to this mode which are covered in more detail below. The other mode is “Advanced.” This mode requires a better understanding of the ADCplate and some extra coding but allows you to configure channels for different sample rates, read large blocks of acquired values, continuously stream data, and trigger measurements.

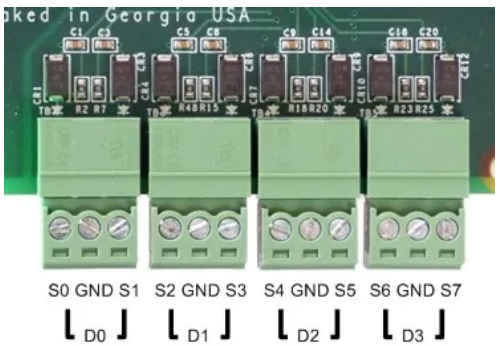
Events

For the ADCplate, the term “events” is somewhat analogous to the “interrupts” and “service requests” used in previous Pi-Plates. However, when using Advanced mode, they are a necessary method of signaling the Raspberry Pi after a lengthy process has completed. For example, when using block mode or streaming at high precision sample rates, it can take many seconds or even minutes before data is available to read. Since it would consume too much bandwidth to just wait for these measurements to complete, the ADCplate asserts a GPIO pin on the Raspberry Pi to signal when the data is ready.

Events can also be used in Easy mode to signal when a “fresh” set of values is available. This is covered in more detail in the next section.

Voltage Input Configuration

The ADCplate can support a combination of eight ground referenced single ended inputs or four, true differential inputs. A differential input is the combination of two single ended inputs. We have attempted to illustrate this in the figure below. Here we see that differential input D0 is the measurement of S0-S1, D1 is S2-S3, and so on.



Easy Mode

Operation

After power up or a reset, the ADCplate defaults to Easy mode. In this mode, the ADCplate continually collects data from all of the single ended voltage inputs (8 values), differential voltage inputs (4 values), and 4-20mA inputs (4 values). This data can be read at any time using either the getADC command for a single channel or the getADCall command for all the channels. Easy, right? In addition, you can select three different Easy modes that tradeoff accuracy for speed. These modes are:

Accuracy/Speed
"HIGH"/SLOW"
"MEDium"/"MEDium"
"LOW"/"FAST"

To change the Easy mode, simply send the `setMODE(addr,mode)` command to the ADCplate where *addr* is the board address and *mode* is *"HIGH", "SLOW", "MED", "LOW", or "FAST"*. The mode arguments can be lower case or upper case. The performance of each of the Easy modes is shown in the table below:

Search:

Easy Mode	Update Rate (Hz)	Cutoff Frequency (Hz)	50Hz Rejection (dB)	60Hz Rejection (dB)	Equivalent Number of Bits Voltage	Equivalent Number of Bits Current	rms Noise (uV / nA)
High Precision Slow Speed	1.25	10	-62	-62	21.6	23.6	6.09 / 7.68
Medium Precision Medium Speed	26.32	174.2	NA	NA	20.2	19.9	17 / 21
Low Precision High Speed	125	6776	NA	NA	17.5	17	106 / 155

Showing 1 to 3 of 3 entries

The values in the "Update Rate" column reflect how frequently the ADCplate updates all of the analog measurements. You should try to avoid reading data faster than the Update Rate because you will likely read the same value more than once. This is called "stale" data. To avoid "stale" data and always read "fresh" data, ensure that you execute your read functions at a frequency less than the Update rate. To ensure that you are always get fresh data at the maximum rate, refer to the Events section below.

Functions

The following read functions are available in Easy mode:

`getADC(addr, input)` – returns the value of the specified analog input. Valid input arguments are 'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7' for single ended measurements, 'D0', 'D1', 'D2', 'D3' for differential measurements, and 'I0', 'I1', 'I2', 'I3' for 4-20mA measurements. The returned values of current measurements will be in units of milliamps.
`getADCall(addr)` – returns a list with all 12 analog input combinations with a format of [S0,S1,S2,S3,S4,S5,S6,S7,D0,D1,D2,D3,I0,I1,I2,I3]
`getSall(addr)` – a convenience function that only returns a list of all 8 single ended voltage inputs with the format [S0,S1,S2,S3,S4,S5,S6,S7]. It does not affect the data acquisition time
`getDall(addr)` – a convenience function that only returns a list of all 4 differential voltage inputs with the format [D0,D1,D2,D3]. It does not affect the data acquisition time
`getIall(addr)` – a convenience function that only returns a list of all 4 current inputs in mA with the format [I0,I1,I2,I3]. It does not affect the data acquisition time

Here is an simple example of collecting 10 sets of readings using Easy mode:

```
import piplates.ADCplate as ADC
import time

print(ADC.getID(0))
for i in range(10):
    print(ADC.getADCall(0))
    time.sleep(0.038)
```

Running the above script produces the following results:

```
pi@raspberrypi:~ $ python easyMode.py
Pi-Plate ADCplate
[10.0013375, 2.3930788, 2.4846822, 2.4843127, 2.5980353, 2.3715436, 2.4842441, 2.4842173, 7.598716, -8.05e-05,
0.2267867, -3.28e-05, 20.0560182, 0.0, 0.0, 0.0]
[10.0013375, 2.3927897, 2.4841368, 2.4841338, 2.598089, 2.371639, 2.4843782, 2.4843097, 7.5986713, -2.68e-05,
```

```
0.2267838, 6e-06, 20.0561255, 0.0, 0.0, 0.0]
[10.0013435, 2.3927093, 2.484557, 2.4847329, 2.598092, 2.3716629, 2.4843663, 2.4842083, 7.5989157, -3.28e-05,
0.2268165, 3.28e-05, 20.0557888, 0.0, 0.0, 0.0]
[10.0013107, 2.3930281, 2.4849683, 2.4846852, 2.5980651, 2.3715734, 2.4841815, 2.4841875, 7.5986594, -2.98e-05,
0.226754, 6e-06, 20.0562477, 0.0, 0.0, 0.0]
[10.0013018, 2.392593, 2.4843782, 2.4840653, 2.5980055, 2.3715824, 2.4843037, 2.4842799, 7.6005042, -3e-06,
0.2268046, -1.79e-05, 20.0561106, 0.0, 0.0, 0.0]
[10.0013405, 2.3926258, 2.484253, 2.4843425, 2.5980532, 2.3716658, 2.484408, 2.484268, 7.5989574, -4.77e-05,
0.2268195, -2.98e-05, 20.056662, 0.0, 0.0, 0.0]
[10.0013405, 2.3929328, 2.4848282, 2.484861, 2.5980771, 2.3716331, 2.4842501, 2.4841428, 7.5988322, 0.0,
0.2267689, -8.9e-06, 20.056349, 0.0, 0.0, 0.0]
[10.0012839, 2.3930192, 2.4847627, 2.4844378, 2.5980264, 2.3715705, 2.484256, 2.4842322, 7.5986773, -5.96e-05,
0.2268106, 6e-06, 20.0562656, 0.0, 0.0, 0.0]
[10.0013763, 2.3927867, 2.4842083, 2.4840683, 2.5979877, 2.3715913, 2.4843752, 2.4842739, 7.5988233, -2.38e-05,
0.2267867, 1.19e-05, 20.0559109, 0.0, 0.0, 0.0]
[10.0012988, 2.3927063, 2.4844795, 2.4846226, 2.598089, 2.371642, 2.4843276, 2.4841875, 7.5989515, -3.28e-05,
0.2268016, 8.9e-06, 20.0566173, 0.0, 0.0, 0.0]
pi@raspberrypi:~ $
```

For the above example, we connected a 10V power supply across S0, a short across D1 and D3, and a 20mA current source into I0. The script collected and printed 10 lists of analog readings using the default MEDium settings in Easy mode. The format of each list is [S0,S1,S2,S3,S4,S5,S6,S7,D0,D1,D2,D3,I0,I1,I2,I3]. Note that if a voltage input is left floating, the ADC measures the bias voltage on an internal resistive divider. This is why there are a series of measurements in the 2.5V range in the data above. If this is undesirable, then you will need to short out unused inputs or only read the input channels with valid inputs using the **`getADC(addr,input)`** command.

Events

When we issue read instructions in Easy mode faster than the Update Rate, we run the risk of reading "stale" data, If we go slower than the Update Rate, we may miss "fresh" data. However, by enabling and monitoring events, it is possible to read data at precisely the Update Rate – no more "stale" data and no missed "fresh" data. We will be using three functions for event monitoring. They are:

`enableEVENTS(addr,signal – optional)` – the ADCplate will pull down the selected GPIO pin if an enabled event occurs. If included, the signal argument can be either `SHARED` (use GPIO22) or `DEDICATED` (more on this later). If the optional signal argument is omitted, the ADCplate will use the last selected signal. At power up or after an `initADC`, this will default to `SHARED`.
`check4EVENTS(addr)` – reads the status of the enabled event pin and returns True if an event has occurred or a False if there is no change.
`getEVENTS(addr)` – this returns the event status register of the addressed ADCplate. This function must be used after every detected event to clear the selected pin. The bits in the events status register are mapped as follows:

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
ADC	NA	NA	NA	DIN3	DIN2	DIN1	DIN0

The following script demonstrates how to use the event functions:

```
import piplates.ADCplate as ADC
import time
addr=0
print(ADC.getID(addr))
ADC.enableEVENTS(addr)   #Enable events and used default of SHARED
ADC.getEVENTS(addr)      #Clear our residual events
for i in range(10):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #2nd while loop to check for events
            pass                               #if no event then pass and check again
        if (ADC.getEVENTS(addr) &amp; 0x80):       #read event register when detected
            go=False                           #if the event is == ADC complete (0x80) then get out of while loop and fetch dat
    print(ADC.getADCall(addr))                 #Collect data and print
```

Advanced Mode

Advanced mode enables a number of powerful functions that allow you to tailor ADCplate data collection to fit your needs. To enter Advanced mode, issue the command `setMODE(addr,'ADV')`. The ADCplate will remain in Advanced mode until another setMode command is issued or a reset or initADC command is issued. Note that events are automatically enabled when the ADCplate enters Advanced mode.

Channel Configuration

In Advanced mode, you can configure each of the analog inputs with a unique sample rate. But, you are limited to a maximum of eight "enabled" inputs. This can be any combination of single-ended, differential, or current inputs as long as the number of enabled inputs does not exceed eight. The ADCplate Python module will issue an error if you attempt to enable too many input channels.

There are three primary functions related to input channel configuration. They are:

`configINPUT(addr,channel,sampleRate,enable – optional)` – this function sets the sample rate (0-18) of the specified analog

channel. See the able below for the ADC characteristics for each Sample Rate. If included, the optional enable argument (True or False) will set the enable status of the channel. If excluded, the status will be unchanged – the default is False. Note that each of the 16 possible analog channels can be configured but only a maximum of eight can be enabled at a time. The channel input arguments are 'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7' for single ended measurements, 'D0', 'D1', 'D2', 'D3' for differential measurements, and 'I0', 'I1', 'I2', 'I3' for 4-20mA measurements. Numeric values 0 through 15 can be used for channel selection as well.

`enableINPUT(addr,input)` – enable a configured input

`disableINPUT(addr,input)` – disable a configured input

Below is a table showing the characteristics of each sample rate. Note how the precision of the measurements improves with slower sample rates. This is a characteristic of the Sigma-Delta A2D converter used on the ADCplate. Also note that when more than channel is enabled, the multiplexer inside the A2D chip requires additional settling time – this reduces the overall throughput. To obtain the fastest sample rate (Fdata in the table), only a single input can be enabled.

Search:

Sample Rate Value	Single Channel Sample Rate (SPS)	Settling Time (msec)	Multi Channel Sample Rate (SPS)	Cutoff Frequency (Hz)	50Hz Rejection (dB)	60Hz Rejection (dB)	Equivalent Number of Bits Voltage	Equivalent Number of Bits Current	rms Noise (uV / nA)
0	1.25	2400	1.25	0.3	-125.63	-130.43	22.7	24	2.9 / 2.4
1	2.5	1200	2.5	0.6	-108.66	-113.49	22.7	24	3.0 / 2.6
2	5	600	5	1.3	-103.09	-107.92	22.4	24	3.5 / 3.1
3	10	300	10	2.6	-101.71	-106.52	22.4	24	3.7 / 3.8
4	16.67	60	16.67	10	-90	-90	21.9	24	5.38 / 7.25
5	20	50	20	10	-85	-85	21.8	24	5.54 / 7.26
6	25	40	25	10	-62	-62	21.6	23.6	6.09 / 7.68
7	50	60	49.68	12.8	-100.76	-46.95	21.8	23.7	5.3 / 7.2
8	59.98	50.02	59.52	15.4	-40.34	-105.8	21.6	23.6	6.2 / 7.6
9	100.2	10	100.2	44	NA	NA	21.3	20.6	8.0 / 13
10	200.3	5	200.3	89.4	NA	NA	20.6	20.1	13 / 18
11	381	2.63	380.95	174.2	NA	NA	20.2	19.9	17 / 21
12	504	1.99	503.8	234	NA	NA	19.9	19.4	21 / 29
13	1007	0.993	1007	502	NA	NA	19.5	18.8	27 / 43
14	2597	0.385	2597	1664	NA	NA	19.7	18	47 / 75
15	5208	0.321	3115	2182	NA	NA	18.3	17.9	62 / 84
16	10415	0.225	4444	3944	NA	NA	17.9	17.4	82 / 113
17	15625	0.193	5181	5164	NA	NA	17.7	17.2	94 / 136
18	31250	0.161	6211	6776	NA	NA	17.5	17	106 / 155

Showing 1 to 19 of 19 entries

In the table above:

- SR: the sample rate index passed to the ADCplate

- Fdata: the sample rate IF ONLY ONE CHANNEL IS ENABLED
- Ts: the settling time in milliseconds
- Fmux: the sample rate if more than one channel is enabled
- fc: the 3dB cutoff frequency of the digital filter internal to the A2D converter
- 50Hz Rej: the amount of attenuation (in dB) at 50 Hz
- 60Hz Rej: the amount of attenuation (in dB) at 60 Hz
- Venb: the equivalent number of voltage measurement bits
- Ienb: the equivalent number of current measurement bits

Note that after entering the Advanced mode, the default input channel sample rate is 10SPS (SR=3) and all channels are disabled.

Finally, there's a helper function in the Python module that prints the key specifications for each sample rate. Just enter srTable() from the command console like this:

```
pi@raspberrypi:~ $ python
Python 3.9.2 (default, Mar 12 2021, 04:06:34)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import piplates.ADCplate as ADC
>>> ADC.srTable()
============================================================
SR Fdata  Ts(msec) Fmux(Hz) fc (Hz)  50Hz Rej 60Hz Rej Venb Ienb
0  1.25    2400     1.25     0.3     -125.63 -130.43   22.7  24
1  2.5     1200     2.5      0.6     -108.66 -113.49   22.7  24
2  5       600      5        1.3     -103.09 -107.92   22.4  24
3  10      300      10       2.6     -101.71 -106.52   22.4  24
4  16.667  60       16.67    ~10     -90     -90       21.9  24
5  20      50       20.00    ~10     -85     -85       21.8  24
6  25      40       25.00    ~10     -62     -62       21.6  23.6
7  50      60       49.68    12.8    -100.76 -46.95    21.8  23.7
8  59.98   50.02    59.52    15.4    -40.34  -105.8    21.6  23.6
9  100.2   10       100.20   44.0      NA      NA      21.3  20.6
10 200.3   5        200.30   89.4      NA      NA      20.6  20.1
11 381     2.63     380.95   174.2     NA      NA      20.2  19.9
12 504     1.99     503.8    234.0     NA      NA      19.9  19.4
13 1007    0.993    1007.00  502       NA      NA      19.5  18.8
14 2597    0.385    2597.00  1664.0    NA      NA      18.7  18.0
15 5208    0.321    3115     2182      NA      NA      18.3  17.9
16 10417   0.225    4444     3944      NA      NA      17.9  17.4
17 15625   0.193    5181     5164      NA      NA      17.7  17.2
18 31250   0.161    6211     6776      NA      NA      17.5  17.0
============================================================
>>>
```

Read Single Channel

The single channel read functions in Advanced mode can be used at any time independently of the state of the other channels. Before executing, these functions save the state (sample rate and enable status) of all input channels. After execution, all of the channel states are restored. The following functions are available for a single channel read:

`readSINGLE(addr,input,sampleRate-optional)` – reads a single analog input even if it is not enabled. If the optional sample rate argument is not included in the function call, the ADCplate will use the existing configuration. This function saves the state of all the channels, performs a measurement, then restores the state of all the channels to their initial configurations.

The **`readSINGLE`** function is actually a combination of two other functions:

`startSINGLE(addr,input,sampleRate-optional)` – starts the measurement of a single analog input even if it is not enabled. When combined with event monitoring and the `getSINGLE` function, this is the recommended method of performing long, high precision measurements. If the optional sample rate argument is not included in the function call, the ADCplate will use the existing configuration. This function does not affect any other preconfigured channels.

`getSINGLE(addr,input)` – returns the value of a single analog input initiated by the `startSINGLE` function.

Care must be taken when using readSINGLE with slow, high precision sample rates since these can cause your code to get stuck in long delays while waiting for the ADCplate to complete a measurement. For example, the script below sets the 'S0' sample rate argument to 0 which requires 2400mSec of settling time for each reading.

```
import piplates.ADCplate as ADC
import time

addr=0

print(ADC.getID(addr))
ADC.setMODE(0,'ADV')
ADC.configINPUT(0,'s0',0)  #Configure S0 input for highest precision
for i in range(10):        #High precision causes BW wasting loop
    t0=time.time()         #Record start time
    print('Index:',i,'S0:',ADC.readSINGLE(addr,'S0'),'Measurement Time:',time.time()-t0)
```

The above script produces the following output:

```
pi@raspberrypi:~ $ python readSingle.py
Pi-Plate ADCplate
```

```
Index: 0 S0: 10.001227 Measurement Time: 2.4031224250793457
Index: 1 S0: 10.001212 Measurement Time: 2.4030823707580566
Index: 2 S0: 10.001165 Measurement Time: 2.4030888080596924
Index: 3 S0: 10.001168 Measurement Time: 2.403085231781006
Index: 4 S0: 10.001141 Measurement Time: 2.403075695037842
Index: 5 S0: 10.001138 Measurement Time: 2.403092861175537
Index: 6 S0: 10.001132 Measurement Time: 2.4030826091766357
Index: 7 S0: 10.001099 Measurement Time: 2.4030978679656982
Index: 8 S0: 10.001096 Measurement Time: 2.4030776023864746
Index: 9 S0: 10.001156 Measurement Time: 2.403125047683716
```

Note how it takes 2.4 seconds for each measurement to complete.

Using `startSINGLE`, event monitoring, and `getSINGLE` allows you to run other tasks while the ADCplate performs a lengthy measurement. The script below shows an example where we make a series of calls to a function to calculate Fibonacci numbers while waiting for the ADCplate to complete a high resolution measurement on a 10V power supply:

```
import piplates.ADCplate as ADC
import time

addr=0

N=0
F0=0
F1=1
FN=0
def Fibonacci():
    global N, F0, F1, FN
    FN=F0+F1
    F0=F1
    F1=FN
    N=N+1
    return N

print(ADC.getID(addr))
ADC.setMODE(0,'ADV')
ADC.configINPUT(0,'s0',0)  #Configure S0 input for highest precision
ADC.enableEVENTS(addr)     #Enable events and used default of SHARED
ADC.getEVENTS(addr)        #Clear out any resual events

for i in range(10):
    ADC.startSINGLE(0,'S0')
    go=True
    Fcount=0
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for event
            #Your code starts here:
            Fcount=Fibonacci()                 #if no event then calculate the next Fibonacci number
            #Your code ends here
        if (ADC.getEVENTS(addr) &amp;amp; 0x80):       #read event register when event detected
            go=False                           #if the event is == ADCcomplete then get out of while loop and fetch data
    print('Channel S0:',ADC.getSINGLE(addr,'S0'),'Total Fibbonacci Values Calculated:',Fcount)
```

The output from the above script looks like:

```
pi@raspberrypi:~ $ python SingleWithEvent.py
Pi-Plate ADCplate
Channel S0: 10.001078 Total Fibonacci Values Calculated: 151035
Channel S0: 10.001069 Total Fibonacci Values Calculated: 225574
Channel S0: 10.001159 Total Fibonacci Values Calculated: 283345
Channel S0: 10.001132 Total Fibonacci Values Calculated: 332209
Channel S0: 10.001147 Total Fibonacci Values Calculated: 375421
Channel S0: 10.00115 Total Fibonacci Values Calculated: 414490
Channel S0: 10.001153 Total Fibonacci Values Calculated: 450472
Channel S0: 10.001144 Total Fibonacci Values Calculated: 483968
Channel S0: 10.001108 Total Fibonacci Values Calculated: 515359
Channel S0: 10.001162 Total Fibonacci Values Calculated: 545064
```

Note the area in the script where you can place your own code that executes while waiting for the ADCplate to finish the long measurement.

Reading a Scan

A scan is a single read of all the enable input channels on the ADCplate. After a scan is read, the Python code returns a list that contains 16 values. The data will be arranged in the list in the following order: [S0,S1,S2,S3,S4,S5,S6,S7,D0,D1,D2,D3,I0,I1,I2,I3]. Channels that have been enabled will be populated with their measured values. Channels that are disabled will say 'None.' The simplest way to read a scan is with the readSCAN function:The following functions are available for reading a scan:

`readSCAN(addr)` – performs a scan and returns a list of all 16 possible analog inputs. The value of each enabled channel will appear in the list. All disabled channels will have have a value of 'None'.

```
import piplates.ADCplate as ADC
```

```
import time

addr=0

print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'S0',10,True)  #Configure S0 input
ADC.configINPUT(addr,'I0',2,True)   #Configure I0 input
ADC.configINPUT(addr,'D3',18,True)  #Configure D3 input
ADC.configINPUT(addr,'D1',18,True)  #Configure D1 input
for i in range(10):
    t0=time.time()         #Record start time
    print('Index:',i,'Scan:',ADC.readSCAN(addr),'Scan Measurement Time:',time.time()-t0)
```

The output of the above script is:

```
pi@raspberrypi:~ $ python readSCAN.py
Pi-Plate ADCplate
Index: 0 Scan: [10.001266, None, None, None, None, None, None, None, -0.000194, None, -4.5e-05,
20.055681, None, None, None] Scan Measurement Time: 0.6124236583709717
Index: 1 Scan: [10.001317, None, None, None, None, None, None, None, -0.000274, None, 9.2e-05, 20.055607,
None, None, None] Scan Measurement Time: 0.6124956607818604
Index: 2 Scan: [10.00126, None, None, None, None, None, None, None, None, -0.000119, None, -0.000235,
20.055598, None, None, None] Scan Measurement Time: 0.6118607521057129
Index: 3 Scan: [10.001287, None, None, None, None, None, None, None, None, 2.4e-05, None, -1.5e-05, 20.055586,
None, None, None] Scan Measurement Time: 0.6118383407592773
Index: 4 Scan: [10.001329, None, None, None, None, None, None, None, None, -0.00017, None, 7.7e-05, 20.055664,
None, None, None] Scan Measurement Time: 0.6118028163909912
Index: 5 Scan: [10.001254, None, None, None, None, None, None, None, None, 5.7e-05, None, -9e-06, 20.055568,
None, None, None] Scan Measurement Time: 0.611842155456543
Index: 6 Scan: [10.001287, None, None, None, None, None, None, None, None, -8e-05, None, -6e-05, 20.055613,
None, None, None] Scan Measurement Time: 0.6118278503417969
Index: 7 Scan: [10.001296, None, None, None, None, None, None, None, -3.6e-05, None, -3.6e-05, 20.055571,
None, None, None] Scan Measurement Time: 0.611842155456543
Index: 8 Scan: [10.001278, None, None, None, None, None, None, None, None, -1.8e-05, None, -0.000134,
20.055577, None, None, None] Scan Measurement Time: 0.6118307113647461
Index: 9 Scan: [10.001311, None, None, None, None, None, None, None, None, -0.000215, None, 6e-05, 20.055643,
None, None, None] Scan Measurement Time: 0.6118578910827637
pi@raspberrypi:~ $
```

The data displayed above shows how the string “None” is used to indicate a disabled channel. Also of note is the scan measurement time. This can become a large value when one are more channels are configured for high accuracy. **readSCAN** is actually a combination of the following two functions. Similar to readSINGLE these should be used with event monitoring to minimize wasted bandwith:

startSCAN(addr) – starts a scan of all of the enabled channels. When combined with event monitoring and the getSCAN function, this is the recommended method of performing high precision measurements on multiple channels.

getSCAN(addr) – returns a list of all 16 possible analog inputs collected by the startSCAN function. The value of each enabled channel will appear in the list. All disabled channels will have have a value of ‘None’

Here’s a script that starts a scan and then calculates the Fibonacci sequence while waiting for the ADC complete event to occur:

```
import piplates.ADCplate as ADC
import time

addr=0

N=0
F0=0
F1=1
FN=0
def Fibonacci():
    global N, F0, F1, FN
    FN=F0+F1
    F0=F1
    F1=FN
    N=N+1
    return N

print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'S0',10,True)
ADC.configINPUT(addr,'I0',1,True)
ADC.configINPUT(addr,'D3',18,True)
ADC.configINPUT(addr,'D1',18,True)
for i in range(10):
    ADC.startSCAN(0)
    go=True
    Fcount=0
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for event
        # Start of foreground tasks
            Fcount=Fibonacci()                 #if no event then calculate the next Fibonacci number
        # End of foreground tasks
        if (ADC.getEVENTS(addr) &amp;amp; 0x80):       #read event register when event detected
            go=False                           #if the event is == ADCcomplete then get out of while loop and fetch data
```

```
    print('Count:',i+1,'Scanned Data:',ADC.getSCAN(addr),'Total Fibonacci Values Calculated:',Fcount)
```

The output from the above was:

```
pi@raspberrypi:~ $ python ScanWithEvent.py
Pi-Plate ADCplate
Count: 1 Scanned Data: [2.363315, None, None, None, None, None, None, None, None, -4.8e-05, None, -1.2e-05,
20.056468, None, None, None] Total Fibonacci Values Calculated: 99188
Count: 2 Scanned Data: [2.363342, None, None, None, None, None, None, None, None, 6.9e-05, None, -5.1e-05,
20.056477, None, None, None] Total Fibonacci Values Calculated: 151077
Count: 3 Scanned Data: [2.363634, None, None, None, None, None, None, None, None, -0.00014, None, -6.3e-05,
20.056453, None, None, None] Total Fibonacci Values Calculated: 191221
Count: 4 Scanned Data: [2.362815, None, None, None, None, None, None, None, None, -0.000137, None, -0.000173,
20.056471, None, None, None] Total Fibonacci Values Calculated: 225370
Count: 5 Scanned Data: [2.362889, None, None, None, None, None, None, None, None, -7.5e-05, None, 4.2e-05,
20.056516, None, None, None] Total Fibonacci Values Calculated: 255653
Count: 6 Scanned Data: [2.363643, None, None, None, None, None, None, None, None, -0.000149, None, 0.000143,
20.056492, None, None, None] Total Fibonacci Values Calculated: 283154
Count: 7 Scanned Data: [2.363122, None, None, None, None, None, None, None, None, -0.000152, None, -0.000218,
20.05648, None, None, None] Total Fibonacci Values Calculated: 308488
Count: 8 Scanned Data: [2.362666, None, None, None, None, None, None, None, None, -1.2e-05, None, -0.000137,
20.056379, None, None, None] Total Fibonacci Values Calculated: 331985
Count: 9 Scanned Data: [2.363452, None, None, None, None, None, None, None, None, -0.000259, None, -2.4e-05,
20.056447, None, None, None] Total Fibonacci Values Calculated: 354177
Count: 10 Scanned Data: [2.363443, None, None, None, None, None, None, None, None, -0.00014, None, -4.2e-05,
20.056465, None, None, None] Total Fibonacci Values Calculated: 375248
```

Reading a Block

The ADCplate can collect data in blocks as large as 8192 readings independently of the Raspberry Pi. This operation requires following two functions:

startBLOCK(addr,num) – initiates a block read of all enabled channels. The *num* argument can range from 1 to 8192. This function must be used with event monitoring and the getBLOCK function. This is the most effective method for high speed measurements of a single channel.

getBLOCK(addr) – returns a list of all of the measurements collected by the **startBLOCK** function. The returned data is a sequential list of all the enabled channels. If more than one channel is enabled then the ADCplate will sequence the data from S0, S1, S2, S3, S4, S5, S6, S7, D0, D1, D2, D3, I0, I1, I2, and I3. For example, if S0, I0, D1, and D3 are enabled for a block read, then the data sequence in the returned list will be : S0, D1, D3, I0, S0, D1, D3, I0, S0, D1, D3, I0 and so.

In the following script, we will do a block read of a single differential channel with a sample rate of 31250 samples per second:

```
import piplates.ADCplate as ADC
import time

addr=0

print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'D0',18,True)  #Configure S0 input for highest speed
#ADC.configINPUT(addr,'I0',18,True)  #Configure I0 input for highest speed
N=2048
block=[0]*N    #initialize the list
for i in range(10):
    ADC.startBLOCK(0,N)  #start the block
    go=True
    t0=time.time()
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for events
        # Start of foreground tasks
            pass
        # End of foreground tasks
        if (ADC.getEVENTS(addr) &amp;amp; 0x80):       #read event register when event detected
            go=False                           #if the event is == ADCcomplete then get out of while loop and fetch data
    block = ADC.getBLOCK(addr)
    print('Count:',i+1,'Block Length:',len(block),'Block Data:',block[0:4],'Time Required:',time.time()-t0)
```

To keep the listing simple we placed a “pass” statement where foreground code would normally go. And to keep the rest of this example short, we only print the first four values of each block:

```
pi@raspberrypi:~ $ python OneChannelBlock.py
Pi-Plate ADCplate
Count: 1 Block Length: 2048 Block Data: [6.00771, 6.007716, 6.007779, 6.00771] Time Required:
0.08788800239562988
Count: 2 Block Length: 2048 Block Data: [6.007522, 6.00771, 6.007865, 6.007814] Time Required:
0.0882728099822998
Count: 3 Block Length: 2048 Block Data: [6.00754, 6.007856, 6.00794, 6.007999] Time Required:
0.08726310729980469
Count: 4 Block Length: 2048 Block Data: [6.007624, 6.007698, 6.007817, 6.007797] Time Required:
0.08809161186218262
```

```
Count: 5 Block Length: 2048 Block Data: [6.007531, 6.007594, 6.007618, 6.007692] Time Required:
0.0883338451385498
Count: 6 Block Length: 2048 Block Data: [6.007701, 6.007692, 6.007811, 6.007901] Time Required:
0.08880352973937988
Count: 7 Block Length: 2048 Block Data: [6.007418, 6.007552, 6.007656, 6.007481] Time Required:
0.08847951889038086
Count: 8 Block Length: 2048 Block Data: [6.007877, 6.007573, 6.007594, 6.007814] Time Required:
0.08814883232116699
Count: 9 Block Length: 2048 Block Data: [6.007668, 6.007648, 6.007576, 6.007737] Time Required:
0.08852410316467285
Count: 10 Block Length: 2048 Block Data: [6.007677, 6.007779, 6.007591, 6.007543] Time Required:
0.08782434463500977
```

One final note: if you calculate the total time to collect 2048 samples at 31,250sps the you get 65.5msec. However, the above data shows around 85msec. The difference is the time required to convert the 24bit values to floating point and the time required to transfer the block data from the ADCplate to the RPi.

In our next example we will enable I0 by uncommenting the config line: **`ADC.configINPUT(addr,'I0',18,True)`**

```
pi@raspberrypi:~ $ python OneChannelBlock.py
Pi-Plate ADCplate
Count: 1 Block Length: 2048 Block Data: [6.00785, 19.989416, 6.007805, 19.991696] Time Required:
0.3614692687988281
Count: 2 Block Length: 2048 Block Data: [6.007752, 19.991463, 6.00785, 19.989896] Time Required:
0.3519153594970703
Count: 3 Block Length: 2048 Block Data: [6.007975, 19.992656, 6.007659, 19.990417] Time Required:
0.35117053985595703
Count: 4 Block Length: 2048 Block Data: [6.007975, 19.992498, 6.007877, 19.992286] Time Required:
0.3515782356262207
Count: 5 Block Length: 2048 Block Data: [6.007946, 19.989592, 6.008103, 19.990775] Time Required:
0.351391077041626
Count: 6 Block Length: 2048 Block Data: [6.007996, 19.992948, 6.007999, 19.991937] Time Required:
0.3514230251312256
Count: 7 Block Length: 2048 Block Data: [6.007963, 19.989383, 6.007987, 19.990093] Time Required:
0.3514888286590576
Count: 8 Block Length: 2048 Block Data: [6.007648, 19.991571, 6.007898, 19.990832] Time Required:
0.351654052734375
Count: 9 Block Length: 2048 Block Data: [6.00777, 19.991404, 6.007937, 19.990465] Time Required:
0.3512444496154785
Count: 10 Block Length: 2048 Block Data: [6.00771, 19.991964, 6.007904, 19.993803] Time Required:
0.351564884185791
```

The block data now shows the alternating values of D0 and I0. In addition, the total time to collect 2048 readings has increased from 85msec to 351msec. This is because every time the multiplexor switches to a different input, it requires a certain amount of time to settle before it can perform the conversion. This affects the overall sample rate which, in this case has been reduced to 6,211SPS.

Streaming

In streaming mode, the ADCplate collects data from each enabled channel indefinitely and places it in a FIFO (First In First Out) buffer. The ADCplate will generate an event when the FIFO is half full. Streaming mode can be used to continuously monitor a process or as a method to collect an arbitrarily large block of data.

The basic flow of initiating and collection streamed data is:

1. Enable inputs
2. Start stream
3. Wait for event
4. When even occurs get the buffered stream data
5. Store and/or act on data
6. GOTO step 3

The streaming functions are:

`startSTREAM(addr,num)` – places the ADCplate in streaming mode. The ADC continuously samples and generates an event whenever the FIFO is half full. The '*num*' argument informs the ADC plate the amount of data in the FIFO that triggers an event. This value can range from 1 to 4096 values. A smaller buffer will produce "fresher" data but requires more overhead. A large buffer will require less overhead since reads of the ADCplate occur less often. Note that startSTREAM must be used with the getSTREAM function and event monitoring.

`getSTREAM(addr)` – returns an N-sized list of streamed values. The number of values is set by the *num* argument in the startSTREAM function. The returned data is a sequential list of all the enabled channels.

`stopSTREAM(addr)` – stops the stream and places the ADCplate in idle mode.

Below is a simple example that uses the above functions to stream 10240 contiguous readings:

```
import piplates.ADCplate as ADC
import time

addr=0
N=1024
data=list()   #initialize the list

print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'s0',18,True)  #Configure S0 input for highest speed
ADC.configINPUT(addr,'I0',18,True)  #Configure I0 input for highest speed
ADC.startSTREAM(addr,N)             #start the STREAM
t0=time.time()
for i in range(10):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for events
        # Start of foreground tasks
            pass
        # End of foreground tasks
        if (ADC.getEVENTS(addr) &amp;amp; 0x80):       #read event register when event detected
            go=False                           #if the event is == ADCcomplete then get out of while loop and fetch data
    data.extend(ADC.getSTREAM(addr))
ADC.stopSTREAM(addr)               #stop the STREAM
print('Number of readings collected:',len(data),'Measurement Time:',time.time()-t0)
print('First 8 values:',data[0:8])
```

The output of the above is:

```
pi@raspberrypi:~ $ python basicStream.py
Pi-Plate ADCplate
Number of readings collected: 10240 Measurement Time: 1.6614093780517578
First 8 values: [6.007805, 20.026991, 6.007326, 20.027798, 6.007588, 20.025557, 6.007379, 20.028439]
```

It's very easy to collect a large amount of data very quickly while streaming. And, while you can keep this data in a long list, you might want to save it to a file instead. The following example does this by saving the collected values to a file using the CSV (Comma Separated Values) format. The CSV format is readable by LibreOffice Calc and excel:

```
import piplates.ADCplate as ADC
import time

addr=0
N=1024
data=N*[]   #initialize the list
lCount = 10

print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'s0',18,True)  #Configure S0 input for highest speed
ADC.configINPUT(addr,'I0',18,True)  #Configure I0 input for highest speed
cCount = 2                          #cCount is the number of enabled channels
f=open('myLog.csv','w')            #create a log file
f.write('Index,Channel 0 Voltage In,Channel 0 Loop Current'+'\n') #Write the header to the file
ADC.startSTREAM(addr,N)             #start the STREAM
t0=time.time()
index=1
for i in range(lCount):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for events
        # Start of foreground tasks
            pass
        # End of foreground tasks
        if (ADC.getEVENTS(addr) &amp;amp; 0x80):       #read event register when event detected
            go=False                           #if the event is == ADCcomplete then get out of while loop and fetch data
    data=ADC.getSTREAM(addr)
    for k in range(N//cCount):
        f.write(str(index)+','+str(data[2*k])+','+str(data[2*k+1])+'\n') #Convert data to a string and write
        index = index + 1
ADC.stopSTREAM(addr)                           #stop the STREAM
f.close()
```

The above script creates a file called myLog.csv and writes 10,240 values to it. Opening the file with a text editor we see the following:

```
Index,Channel 0 Voltage In,Channel 0 Loop Current
1,6.007707,20.027736
2,6.007645,20.027232
3,6.007883,20.028049
4,6.00782,20.027491
5,6.007674,20.026824
6,6.007853,20.027804
7,6.007683,20.027325
.
.
.
5120,6.007695,20.027494
```

Using the code above, we connected a function generator with a 4Hz stairway signal to S0 and a 6Hz sinc signal to S1. Then, we collected 5120 samples from each (2*5120=10,240), opened the CSV file with LibreOffice Calc, and plotted this data:

Trigger Mode

The ADCplate includes a set of functions that allow Scan measurements to be initiated either with an internal clock, a software command, or with an external input. With triggering, it is possible to:

- Start an ADC conversion within 5 microseconds of an external event
- Synchronize the ADC conversion of multiple ADC plates
- Sample data at programmable rates between 0 and 1Khz
- Start a conversion based on the value of an analog measurement

The functions available for trigger mode include:

`configTRIG(addr,mode,primary – optional)` – configures the trigger mode of the ADCplate at the assigned address. Modes include:

'EXT' – the scan will start when a low to high pulse occurs on the trigger pin.
'SW' – the scan starts when the swTRIGGER function is called
'CLOCKED' – an internal clock triggers scans at a rate configured by the triggerFREQ function.
'OFF' – disables the trigger mode.

If the optional primary argument is set to True, the ADCplate will output a clock signal to the trigger terminal when in 'CLOCKED' mode or a pulse when in 'SW' mode. Setting the primary argument to True is only valid in CLOCKED and SW modes.

`startTRIG(addr)` – places the addressed ADCplate in trigger mode. A call to configTRIG must occur before calling this function. Once trigger mode has been initiated, a combination of event monitoring and getSCAN calls should be used to read the data collected by the ADCplate.

`stopTRIG(addr)` – stops trigger mode on the addressed ADCplate.

`triggerFREQ(addr,freq)` – sets the internally generated trigger clock rate and returns the actual value set (the clock has a limited 16-bit resolution). The frequency should be greater than zero and less than 1000Hz. The user must be careful not to clock faster than scans can be performed.

`maxTRIGfreq(addr)` – a helper function that returns an approximate value of the highest possible trigger rate based on the currently enabled channels.

`swTRIGGER(addr)` – if configured in 'SW' mode, this instructs the addressed ADCplate to immediately trigger a scan. If the ADCplate is the trigger primary, a 1-2msec pulse will be issued on the trigger terminal to signal other ADCplates

When using an external trigger, connect a digital signal to the TRIG terminals on the left side of the ADCplate (see picture below). This pin has a 100K resistor to ground and tolerates voltages as high as 30 volts so it is compatible with PNP switching in an industrial application. And since the switching voltage is 2.4V, it will work with digital logic as well. If multiple ADCplates are on the stack, they can share a common trigger input and be synchronized within 5usec of each other. Or each ADCplate can receive a unique trigger.

Here is an example where the trigger is enabled in EXT mode and a digital signal is applied to the TRIG input:

```
import piplates.ADCplate as ADC
import time

addr=0
tData=list()
N=10
#ADC.initADC(addr)
print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'I0',12,True)  #Configure I0 input
ADC.enableEVENTS(addr)              #Enable events and used default of SHARED
ADC.configTRIG(addr,'EXT')

print('Max trigger rate:',ADC.maxTRIGfreq(addr))
ADC.getEVENTS(addr)      #Clear our residual events
```

```
ADC.startTRIG(addr)
t0=time.time()
for i in range(N):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for event
            pass                               #if no even then check again
        if (ADC.getEVENTS(addr) &amp; 0x80):       #read event register when event detected
            go=False                           #if the event is = ADCcomplete then get out of while loop and fetch data
    tData.append(ADC.getSCAN(addr))
    t1=time.time()
    dt=t1-t0
    t0=t1
    print('External Trigger Frequency:',1/dt)
ADC.stopTRIG(addr)
print('Sample Value:',tData[N-1])
```

The output from the above with a 100Hz trigger clock looks like:

```
pi@raspberrypi:~ $ python externalTRIGGER.py
Pi-Plate ADCplate
Max trigger rate: 504.0
External Trigger Frequency: 129.46981108778863
External Trigger Frequency: 106.51118616521497
External Trigger Frequency: 100.14335171788076
External Trigger Frequency: 99.5940542337465
External Trigger Frequency: 99.54914200270572
External Trigger Frequency: 100.52497363627648
External Trigger Frequency: 100.10988853617204
External Trigger Frequency: 99.95005242588886
External Trigger Frequency: 97.37890044576523
External Trigger Frequency: 103.10481809242872
Sample Value: [None, None, None, None, None, None, None, None, None, None, None, None, 20.026341, None, None,
None]
```

Recall that the output of a scan read is a list showing all possible analog inputs. For our example, only the I0 input is enabled. And, note in the above data that the timing in the first two readings is a bit off due to the way the code first starts out.

Let's repeat the above example but with an internal 100Hz clock signal:

```
import piplates.ADCplate as ADC
import time

addr=0
tData=list()
N=10
#ADC.initADC(addr)
print(ADC.getID(addr))
ADC.setMODE(addr,'ADV')
ADC.configINPUT(addr,'I0',12,True)  #Configure I0 input
ADC.enableEVENTS(addr)              #Enable events and used default of SHARED
ADC.configTRIG(addr,'CLOCKED',True)
print('Set Trigger Frequency:',ADC.triggerFREQ(addr,100))
print('Max trigger rate:',ADC.maxTRIGfreq(addr))
ADC.getEVENTS(addr)      #Clear our residual events
ADC.startTRIG(addr)
t0=time.time()
for i in range(N):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for event
            pass                               #if no even then check again
        if (ADC.getEVENTS(addr) &amp; 0x80):       #read event register when event detected
            go=False                           #if the event is = ADCcomplete then get out of while loop and fetch data
    tData.append(ADC.getSCAN(addr))
    t1=time.time()
    dt=t1-t0
    t0=t1
    print('External Trigger Frequency:',1/dt)
ADC.stopTRIG(addr)
print('Sample Value:',tData[N-1])
```

And again the output:

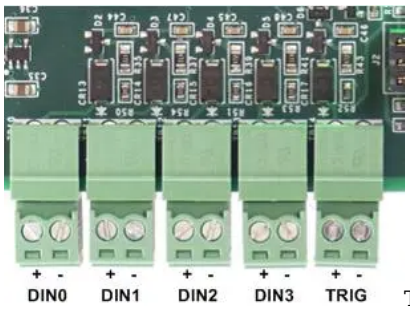
```
pi@raspberrypi:~ $ python clockedTRIGGER.py
Pi-Plate ADCplate
Set Trigger Frequency: 100.0
Max trigger rate: 504.0
External Trigger Frequency: 298.187402246552
External Trigger Frequency: 88.8134502181002
External Trigger Frequency: 103.83996831055654
External Trigger Frequency: 97.75793031115255
External Trigger Frequency: 102.6958523088977
External Trigger Frequency: 100.22471265741117
External Trigger Frequency: 100.1600916993027
External Trigger Frequency: 100.36621201244317
External Trigger Frequency: 98.58280449395949
```

```
External Trigger Frequency: 100.67215514965316
Sample Value: [None, None, None, None, None, None, None, None, None, None, None, None, 20.026779, None, None,
None]
```

The above data is very similar to the previous configuration. And, if you have an oscilloscope or a frequency counter, you will be able to observe a 100Hz square wave on the TRIG connector terminals.

Digital Inputs

The ADCplate includes four digital inputs on the left side for general purpose signal monitoring. These signals are tolerant to input voltages as high as 30V but will also work with signals from digital logic as low as 3.3 volts. The have 33V transient protection and 10K resistors connected to ground making them ideal for PNP switching in an industrial environment. They can be read asynchronously and can also generate events. Note that the TRIG input can also be use as general purpose input as long as the ADCplate is not configured as the primary in trigger mode.



The following functions are provided for these inputs:

getDINbit(addr,bit) – returns the value of the digital input on the specified bit input. Allowable bit values are 0-4. The returned value is 1 for a high input and 0 for a low input.

getDINall(addr) – returns all of the digital inputs as an 8-bit number. Bit 0 is the least significant bit (LSB) and bit 7 is the most significant bit:

```
_____________________
|NA|NA|NA|4|3|2|1|0|
_____________________
```

enableDINevent(addr, bit) – enables the specific digital input to generate an event on a low to high transition. Allowable bit values are 0-3.

disableDINevent(addr, bit) – disables the specified digital input from generating events. Allowable bit values are 0-3.

The above “get” functions can issued at any time and are independent of ADC operations. But, to avoid too much traffic on the signals used to control the Pi-Plates, it is also possible to monitor the D0 through D3 inputs using events:

```
import piplates.ADCplate as ADC
import time

addr=0
N=10
print(ADC.getID(addr))
ADC.enableEVENTS(addr)              #Enable events and used default of SHARED
ADC.enableDINevent(addr,0)
ADC.getEVENTS(addr)                 #Clear our residual events
for i in range(N):
    go=True
    while(go):
        while(ADC.check4EVENTS(addr)!=True):   #Check for event
            pass
            #YOUR CODE HERE
        eV=ADC.getEVENTS(addr) &amp;amp; 0x0F
        if (eV):                        #read event register when event detected
            go=False                    #if the event is a digital input then get out of while loop
    print('Event on DIN0 Detected at t=',time.asctime())
```

In this example we enable DIN0 to generate an event when a low to high transition occurs. Once detected, we print out the time. Below is the output when a 1Hz clock is applied to DIN0:

```
pi@raspberrypi:~ $ python digiEVENT.py
Pi-Plate ADCplate
Event on DIN0 Detected at t= Sat Mar 26 08:36:48 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:49 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:50 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:51 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:52 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:53 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:54 2022
```

```
Event on DIN0 Detected at t= Sat Mar 26 08:36:55 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:56 2022
Event on DIN0 Detected at t= Sat Mar 26 08:36:57 2022
```

LED Control

The ADCplate has a single, green LED that normally reflects the power-on state. However, it is possible to control it using the following commands:

setLED(addr) – turn on the LED
clrLED(addr) – turn off the LED
toggleLED(addr) – if LED is on, turn off. If LED is off, turn on.

System Functions

The ADCplate supports the following common functions:

help() – returns a concise list of the ADCplate functions along with explanations
getID(addr) – return Pi-Plate descriptor string, "Pi-Plates ADCplate"
getFWrev(addr) – return FW revision in decimal format
getHWrev(addr) – return HW revision in decimal format
getVersion() – returns revision of python module
getADDR(addr) – returns address of pi-plate. Used for polling available boards at power up.
RESET(addr) – set ADCplate to power on state.

You have no choice but to operate in a world shaped by globalization and the information revolution. There are two options: adapt or die. Andy Grove

^Back to Top