Monsoon Solutions Inc.

Power Monitor Python API Documentation Version 1.0

Table of Contents

Introduction:

The Monsoon python project is an open source Python implementation of the Monsoon Power Monitor protocol. This project supports two hardware versions, the Low-Voltage Power Monitor (LVPM, Part number FTA22J, has a white case) and the High-Voltage Power Monitor (HVPM, Part number AAA10F, has a black case).

This document solely covers information related to the python implementation of the Power Monitor API. For hardware information and GUI instructions, refer to the Power Monitor End User Manual. For instructions about the use of the C# GUI, refer to the Power Monitor Developer API Guide. Both documents are available at http://msoon.github.io/powermonitor/

Compatibility:

This script has been tested on Windows 10, RHEL 7.3, and macOS Sierra, 10.12.2, using Python 2.7 and Python 3.5.

During development, we have found that pure Python is often not fast enough to keep up with real-time sampling, and lacks the multi-threading capabilities that would be necessary to properly fix this. Users may find large numbers of measurements being dropped during sampling. We are continuing development on a solution that maintains universal compatibility and is fast enough to allow measurements to be taken in real time. Currently, the only workarounds are to use a workstation with a higher single-core clock speed, or to collect and then process measurements separately.

Installation Instructions:

Using pip

Use the command:

‘Pip install monsoon’

Using the installer

Download the Python installer from http://msoon.github.io/powermonitor/ - unzip the contents and find setup.py. From there, use the command:

‘Python setup.py install’

Dependencies:

The following Python libraries are used with this library, and will need to be installed before use.

Numpy: http://www.numpy.org/
or install using 'pip install numpy'

pyUSB: https://github.com/walac/pyusb

or install using 'pip install pyusb'

libusb 1.0: http://www.libusb.org/wiki/libusb-1.0
or install using 'pip install libusb1'

Note: pyUSB also supports libusb 0.1 and OpenUSB as backends, but those haven't been tested with this script and are not officially supported by Monsoon

Preparing your environment:

1. On windows, for any device to be detected by libusb, you will need to install a libusb filter. This can be downloaded from https://sourceforge.net/projects/libusb-win32/. This step can be skipped for Linux and MacOS users.
2. For LVPM users, the firmware on your device will probably not be compatible with the script. Older firmware uses a serial port emulator to communicate with the PC, while the newer firmware uses a full USB interface. A firmware update is available in the /firmware folder of the source package (for pip users, this will also be present in %python%/Lib/site-packages/Monsoon/Firmware. See the reflash example later in this document for instructions on how to reflash your unit's firmware.

Examples:

Begin by importing the Monsoon class. The device you instantiate will depend on what device you have. The easiest way to tell Power Monitors apart at a glance is the color of the case. A LVPM will have a white case, while an HVPM will have a black case. For this example, we'll be assuming you have an LVPM.

Sampling from the main channel:

Create the object appropriate to your Power Monitor hardware, and then call setup_usb(), which will connect to the first available device.

```
import monsoon.LVPM as LVPM
import monsoon.sampleEngine as sampleEngine
import monsoon.Operations as op


Mon = LVPM.Monsoon()
Mon.setup_usb()
```

Next, create an instance of the sampleEngine class. Main channel current and voltage are enabled by default, so there's no need to enable them. The sample engine class defaults to saving samples as a python list that can be retrieved with the getSamples() function. However, if you want to use the built-in CSV output, you need to enable that before sampling starts.

CSV Format outputs a first row of headers indicating the channel and unit of each column. It then outputs one row for each measurement. This is documented in more detail in the sampleEngine class documentation section.

```
Mon.setVout(4.0)
engine = sampleEngine.SampleEngine(Mon)
engine.enableCSVOutput("Main Example.csv")
engine.ConsoleOutput(True)
numSamples=5000 #sample for one second
engine.startSampling(numSamples)
```

After one second, the program should be finished, and you'll have one second worth of samples collected by the Power Monitor. If you want the Power Monitor to continue sampling indefinitely, use the value `sampleEngine.triggers.SAMPLECOUNT_INFINITE` as the numSamples parameter.

Sampling from the USB and Aux Channels:

Create the PM and sample engine the same way as the main channel. Disable the main channels to avoid extra measurements, and then enable the USB and Aux channels:

```
#Disable Main channels
engine.disableChannel(sampleEngine.channels.MainCurrent)
engine.disableChannel(sampleEngine.channels.MainVoltage)
```

```
#Enable USB channels
engine.enableChannel(sampleEngine.channels.USBCurrent)
engine.enableChannel(sampleEngine.channels.USBVoltage)
#Enable Aux channel
engine.enableChannel(sampleEngine.channels.AuxCurrent)
#Set USB Passthrough mode to 'on,' since it defaults to 'auto' and will turn off when 
sampling mode begins.
Mon.setUSBPassthroughMode(op.USB_Passthrough.On)
```

Then you can just start sampling as before:

```
engine.enableCSVOutput("USB Example.csv")
engine.startSampling(numSamples)
```

Using Triggers:

The Power Monitor can be set to trigger on the value in any given channel. The sample engine evaluates all measurements, begins recording measurements when the 'start' trigger condition is met, and stops sampling completely when the 'stop' trigger condition is met. Triggers are set with the setStartTrigger(triggerStyle,triggerValue), and setStopTrigger(triggerStyle,triggerValue) functions, in combination with the setTriggerChannel(Channel) function.

Valid trigger styles are in the sampleEngine.triggers class, and value is context-sensitive based on the trigger channel selected. The valid channels and their units are:

Channel	Unit
timeStamp	seconds
MainCurrent	mA
USBCurrent	mA
AuxCurrent	mA
MainVoltage	V
USBVoltage	V

For example, if you want to begin collecting data when the main channel exceeds 100 mA, and stop sampling when the main channel drops below 10 mA, you would use the following code:

```
#Don't stop based on sample count, continue until the trigger conditions have been 
satisfied.
numSamples=sampleEngine.triggers.SAMPLECOUNT_INFINITE
#Start when we exceed 100 mA
engine.setStartTrigger(sampleEngine.triggers.GREATER_THAN,100)
#Stop when we drop below 10 mA.
engine.setStopTrigger(sampleEngine.triggers.LESS_THAN,10)
#Start and stop judged by the main channel.
engine.setTriggerChannel(sampleEngine.channels.MainCurrent)
#Start sampling.
engine.startSampling(numSamples)
```

The sample engine evaluates the conditions for the start/stop triggers once per batch of measurements. The number of measurements per batch is set by the bulkProcessRate parameter, and defaults to 128 samples. Because of this, a number of measurements immediately before the trigger, and immediately after the trigger, will be stored.

Getting samples back as a Python list instead of a CSV output:

During testing, we found that Python isn't fast enough on some systems to request and process measurements every 200 us, and some samples would be dropped. This approach serves as a workaround, storing all samples in a Python list instead of constantly writing to file is much faster, but it will eventually cause a memory overflow error.

Control over when this occurs can be adjusted by adjusting how many channels are enabled, and adjusting the sampling granularity. With all channels enabled, and 1:1 sampling granularity, an overflow error will occur after about 10 hours.

```
engine.disableCSVOutput()
engine.startSampling(5000)
samples = engine.getSamples()

#Samples are stored in order, indexed sampleEngine.channels values
for i in range(len(samples[sampleEngine.channels.timeStamp])):
    timeStamp = samples[sampleEngine.channels.timeStamp][i]
    Current = samples[sampleEngine.channels.MainCurrent][i]
    print("Main current at time " + repr(timeStamp) + " is: " + repr(Current) + "mA")
```

Reflashing Firmware:

1. On the front panel of the Power Monitor is a small button. The text below it reads "Output enabled." Hold this button in and push the power button. Device should power on, and the LED beside the power button should be amber.

2. Select the new firmware file based on your requirements. Most units already have PM_RevD_Prot17_Ver20.hex flashed at the factory, and will be upgrading to LVPM_RevE_Prot1_Ver21_Beta.fwm

3. Create a Python script to use the reflash class. An example is provided in reflashMain.py:

```
import monsoon.reflash as reflash

Mon = reflash.bootloaderMonsoon()
Mon.setup_usb()
```

```
Header, Hex = Mon.getHeaderFromFWM('LVPM_RevE_Prot1_Ver21_Beta.fwm')
if(Mon.verifyHeader(Header)):
    Mon.writeFlash(Hex)
```

4. Note that .fwm files have a header indicating the hardware compatibility for each firmware file. Previous releases use .hex format, so rolling back to older firmware will skip the verification step. Example code below:

```
Mon = reflash.bootloaderMonsoon()
Mon.setup_usb()
Hex = Mon.getHexFile('PM_RevD_Prot17_Ver20.hex')
Mon.writeFlash(Hex)
```

calibrationData.py

Class calibrationData(calsToKeep=10)

A data structure that keeps track of real-time calibration data sent by the Power Monitor

calsToKeep: The number of calibration samples that should be kept when computing the average calibration value.

clear()

Remove all calibration data.

getRefCal(Coarse)

Returns the average value of the last X reference calibration measurements.
returns coarse measurements if Coarse=True, fine measurements otherwise.

getZeroCal(Coarse)

Returns the average value of the last X zero calibration measurements.
returns coarse measurements if Coarse=True, fine measurements otherwise.

addRefCal(value, Coarse)

Adds value to the rolling Coarse(if Coarse=true) or fine(if Coarse=false) reference calibration average.

addZeroCal(value, Coarse)

Adds value to the rolling Coarse(if Coarse=true) or fine(if Coarse=false) zero calibration average.

HVPM.py and LVPM.py

Classes that represent the Power Monitor hardware.

Use LVPM if you have a white Power Monitor, part number FTA22D
Use HVPM if you have a black Power Monitor, part number AAA10F

Fields:

statusPacket: A Data structure from the Operations statuspacket class. Contains all calibration and diagnostic data retrieved from the Power Monitor EEPROM.

fineThreshold: The measurement level of the fine channel measurements where the Power Monitor will switch over from reporting fine samples to reporting coarse samples. On the LVPM, measurements range from 0-32767, and the default fine threshold is 30000. On the HVPM, measurements range from 0-65535, and the default threshold is 64000.

auxFineThereshold: Similar to fine threshold, but for the Aux channel. Aux measurements always range from 0-32767.

mainVoltageScale: Compensates for the voltage divider at the input to the ADC. LVPM = 2, HVPM = 4.

usbVoltageScale: Compensates for the voltage divider at the input to the ADC. Value is 2 for both LVPM and HVPM.

ADCRatio: Conversion ratio to turn raw ADC measurements (0-65535) to voltage measurements.

factoryRes: Center value for calibration values from the factory, used on main and USB channels.

auxFactoryRes: Center value for calibration values from the factory, used on the Aux channel.

setVout(value):

Set main output voltage in 0.01V increments. Valid values are:
LVPM: 2.01-4.55
HVPM: 0.8-13.5
For both devices, a value of '0' is valid, and turns the voltage off.

setPowerupTime(value):

time in ms where powerupcurrentlimit applies. After this time, runtimecurrentlimit applies.

setPowerUpCurrentLimit(value):

Sets power up current limit. Valid values:
LVPM: 0-8
HVPM: 0-15

setRunTimeCurrentLimit(value):

Sets power up current limit. Valid values:
LVPM: 0-8
HVPM: 0-15

setUSBPassthroughMode(USBPassthroughCode):

USB Passthrough mode. 0 = off, 1 = on, 2 = auto

setVoltageChannel(VoltageChannelCode):

Sets voltage measurement channel. 0 = Main & USB, 1 = Main & Aux

getVoltageChannel():

Returns the voltage channel configuration that will be reported back in sample packets.
0 = Main & USB, 1 = Main & Aux

fillStatusPacket():

Populate the LVPM fields with calibration value stored on the device EEPROM.

StartSampling(calTime=1250,maxTime=0xFFFFFFFF):

Cause the Power Monitor to enter sample mode. When in sample mode, it will take measurements every 200us, and store them in a 16-measurement queue. Measurements can be retrieved up to three at a time with the getBulkData command.
calTime is the time in ms between calibration measurements. Smaller values will produce quicker reaction times in response to rapidly changing current, while larger values will result in fewer measurements being lost to devote to calibration.
maxTime is the number of samples that will be taken before the Power Monitor exits sample mode automatically.

stopSampling():

Cause the Power Monitor to exit sample mode.

BulkRead():

When the Power Monitor is in sample mode, this will return one sample packet. The Power Monitor will return a sample packet of length 21-62, and the array is then padded with zeroes to ensure a constant length packet.

Packet Format:

Table 1: Bulk Packet Structure

Offset	Field	Size	Value	Description
0	droppedCount	2	Count	Number of samples dropped
2	flags	1	Bits	Each bit in the byte corresponds to a status flag. D3-0: sequence number (0-15, increments with each packet), D4: 1 indicates overcurrent or thermal kill, 0 = no error. D5: Main Output, 1 indicates unit is at voltage, 0 indicates output disabled. D6 and 7 are reserved.
3	numObs	1	byte	indicates number of measurements in this packet. Valid values 1-3
4	measurement0	18	structure	Measurement structure, see below.
22	measurement1	18	structure	Measurement structure, if present
44	measurement2	18	structure	Measurement structure, if present

Each measurement consists of a measurement data structure:

Table 2: HVPM Measurement Structure

Offset	Field	Size	Format	Description
0	MainCoarse	2	UInt16	Calibration or measurement value.
2	MainFine	2	UInt16	Calibration or measurement value.
4	USBCoarse	2	UInt16	Calibration or measurement value.
6	USBFine	2	UInt16	Calibration or measurement value.
8	AuxCoarse	2	SInt16	Calibration or measurement value.
10	AuxFine	2	SInt16	Calibration or measurement value.
12	Main Voltage	2	UInt16	Main Voltage measurement, or Aux voltage measurement if setVoltageChannel = 1
14	USB Voltage	2	UInt16	USB Voltage
16	Main Gain	1		Measurement gain control.
17	USB Gain	1		Measurement gain control.

LVPM units use a different measurement architecture, and return signed measurements as a result:

Table 3: LVPM Measurement Structure

Offset	Field	Size	Format	Description
0	MainCoarse	2	SInt16	Calibration or measurement value.
2	MainFine	2	SInt16	Calibration or measurement value.
4	USBCoarse	2	SInt16	Calibration or measurement value.
6	USBFine	2	SInt16	Calibration or measurement value.
8	AuxCoarse	2	SInt16	Calibration or measurement value.
10	AuxFine	2	SInt16	Calibration or measurement value.
12	Main Voltage	2	UInt16	Main Voltage measurement, or Aux voltage measurement if setVoltageChannel = 1
14	USB Voltage	2	UInt16	USB Voltage
16	Main Gain	1		Measurement gain control.
17	USB Gain	1		Measurement gain control.

swizzlePacket(packet):

The endianness of PIC measurements are opposite the endianness of x86 processors. This function swaps the endianness of the packet for easier processing.

Operations.py

class OpCodes:

USB Control Transfer operation codes.

Use pmapi.sendCommand(opcode, value) to set a value, and getValue(opcode, valueLength) to set a value

Table 4: USB Control Transfer operation codes

Operation	Opcode	Format	Description
setMainFineResistorOffset	0x02	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
setMainCoarseResistorOffset	0x11	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
setUsbFineResistorOffset	0x0D	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
setUsbCoarseResistorOffset	0X12	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
setAuxFineResistorOffset	0X0E	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
setAuxCoarseResistorOffset	0X13	1 byte, signed, ohms = 0.05 + 0.0001*value	LVPM Calibration value.
calibrateMainVoltage	0X03	NA, value ignored	Internal voltage calibration, affects accuracy of setMainVoltage
resetPowerMonitor	0X05	NA, value ignored	Reset the PIC. Causes disconnect.
setPowerupTime	0X0C	1 byte, signed	time in milliseconds that the powerup current limit is in effect.
setTemperatureLimit	0X29	2 bytes, Signed Q7.8 format	sets the fan turn-on temperature.
setUsbPassthroughMode	0X10	1 byte. Off = 0, On = 1, Auto = 2	Sets USB Passthrough Mode

setMainFineScale	0X1A	4 bytes, unsigned	HVPM Calibration value
setMainCoarseScale	0X1B	4 bytes, unsigned	HVPM Calibration value
setUSBFineScale	0X1C	4 bytes, unsigned	HVPM Calibration value
setUSBCoarseScale	0X1D	4 bytes, unsigned	HVPM Calibration value
setAuxFineScale	0X1E	4 bytes, unsigned	HVPM Calibration value
setAuxCoarseScale	0X1F	4 bytes, unsigned	HVPM Calibration value
setVoltageChannel	0X23	1 byte	0 = Main & USB voltage measurement, 1 = Main & Aux Voltage Measurements
SetPowerUpCurrentLimit	0X43	2 bytes, HV Amps = 15.625*(1.0-powerupCurrentLimit/65535) LV amps = 8.0*(1.0-powerupCurrentLimit/1023.0)	Current limit from power on until setPowerupTime
SetRunCurrentLimit	0X44	2 bytes, HV Amps = 15.625*(1.0-powerupCurrentLimit/65535) LV amps = 8.0*(1.0-powerupCurrentLimit/1023.0)	current limit from setPowerUpTime until power off.
setMainVoltage	0X41	4 bytes, voltage = value / 1048576	set and enable output voltage
SetMainFineZeroOffset	0X42	4 bytes, signed.	Zero-level offset correction. Used exclusivly in HVPM
SetMainCoarseZeroOffset	0X25	4 bytes, signed.	Zero-level offset correction. Used exclusivly in HVPM
SetUSBFineZeroOffset	0X26	4 bytes, signed.	Zero-level offset correction. Used exclusivly in HVPM
SetUSBCoarseZeroOffset	0X27	4 bytes, signed.	Zero-level offset correction. Used exclusivly in HVPM
FirmwareVersion	0X28	1 byte, unsigned	Read-only, firmware version number.

ProtocolVersion	0XC0	1 byte, unsigned	Read-only, protocol version number
HardwareModel	0XC1	1 byte, unsigned	Read-only. 0 = unknown, 1 = LVPM, 2 = HVPM.

class Control_Codes:

USB Protocol codes, used when sending control transfers.

USB_IN_PACKET
Part of the USB Protocol, indicates a vendor type control transfer from the 'in' endpoint.
USB_OUT_PACKET
Part of the USB protocol, indicates a vendor type control transfer from the 'out' endpoint.
USB_SET_VALUE
Control Transfer command instructing the Power Monitor to either get or set an EEPROM value.
USB_REQUEST_START
Control transfer command instructing the Power Monitor to enter sample mode.
USB_REQUEST_STOP
Control transfer command instructing the Power Monitor to exit sample mode.

class Conversion:

Values used for converting from desktop to the PIC

FLOAT_TO_INT

Used in SetVout(). Multiply the voltage desired (e.g. 3.6V) by this value to produce the number understood by the PIC to represent that voltage.

class USB_Passthrough:

Values for setting or retrieving the USB Passthrough mode.
Off = 0
On = 1
Auto = 2

class VoltageChannel:

Values for setting or retrieving the Voltage Channel.
Main = 0
USB = 1
Aux = 2

class statusPacket:

Values stored in the Power Monitor EEPROM. Each corresponds to an opcode

firmwareVersion:
Firmware version number.
protocolVersion:
Protocol version number.
temperature:
Current temperature reading from the board.
serialNumber:
Unit's serial number.
powerupCurrentLimit:
Max current during startup before overcurrent protection circuit activates. LVPM is 0-8A, HVPM is 0-15A.
runtimeCurrentLimit:
Max current during runtime before overcurrent protection circuit activates. LVPM is 0-8A, HVPM is 0-15A.
powerupTime:
Time in ms the powerupcurrent limit will be used.
temperatureLimit:
Temperature limit in Signed Q7.8 format
usbPassthroughMode:
Off = 0, On = 1, Auto = 2

Scales
Scale values are 32-bit scaling values for the HVPM. They are the scale used to convert raw ADC counts into current values.

mainFineScale:
mainCoarseScale:
usbFineScale:
usbCoarseScale:
auxFineScale:
auxCoarseScale:

Zero Offsets:
Zero Offsets are 32-bit offset values added to all HVPM measurements. This corrects for a small offset at the hardware level that would otherwise interfere with calibration.

mainFineZeroOffset:
mainCoarseZeroOffset:
usbFineZeroOffset:
usbCoarseZeroOffset:

Resistor Offsets
Resistor offsets are the LVPM Calibration value. Represents the sense resistor in current calculation.
ohms = 0.05 + 0.0001*offset

mainFineResistorOffset:
mainCoarseResistorOffset:
usbFineResistorOffset:
usbCoarseResistorOffset:
auxFineResistorOffset:
auxCoarseResistorOffset:

class BootloaderCommands:

Bootloader opcodes. Used when reflashing the Power Monitor
ReadVersion = 0x00
ReadFlash = 0x01
WriteFlash = 0x02
EraseFlash = 0x03
ReadEEPROM = 0x04
WriteEEPROM = 0x05
ReadConfig = 0x06
WriteConfig = 0x07
Reset = 0xFF

class BootloaderMemoryRegions:

Memory regions of the PIC18F4550
Flash = 0x00
IDLocs = 0x20
Config = 0x30
EEPROM = 0xf0

class hexLineType:

Line types used in the intel hex format. Used when reflashing the Power Monitor.
Data = 0
EndOfFile = 1
ExtendedSegmentAddress = 2
StartSegmentAddress = 3
ExtendedLinearAddress = 4
StartLinearAddress = 5

class SampleType(object):

Corresponds to the sampletype field from a sample packet.
Measurement = 0x00
ZeroCal = 0x10
invalid = 0x20
refCal = 0x3

Pmapi.py

Class USB_protcol:

Currently the only officially supported protocol.

sendCommand(operation, value):

Send a USB Control transfer. Normally this is used to set an EEPROM value.
operation is from Operations.OpCodes
value is dependent on the operation.

stopSampling():

Send a control transfer instructing the Power Monitor to stop sampling.

startSampling(calTime, maxTime):

Instruct the Power Monitor to enter sample mode.
calTime is the time in ms between calibration measurements. Smaller values will produce quicker reaction times in response to rapidly changing current, while larger values will result in fewer measurements being lost to devote to calibration.
maxTime is the number of samples that will be taken before the Power Monitor exits sample mode automatically.

getValue(operation,valueLength):

Get an EEPROM value from the Power Monitor.
operation is from Operations.OpCodes
valueLength is the number of bytes we expect to read from the EEPROM.

Reflash.py

class bootloaderMonsoon(object):

setup_usb():

Sets up the USB connection. Searches for a connected USB device with the appropriate PID and VID and stores it inside the bootloaderMonsoon object.

writeFlash(hex):

Writes a hex file to the Power Monitor's PIC.
Input is the hex file in the format returned by getHeaderFromFWM or getHexFile.

getHeaderFromFWM(filename):

Strips the header from a Monsoon FWM file, returns the HEX file and the formatted header.

Returns: headers,hex

getHexFile(filename):

Reads an Intel HEX file and returns it in a format that can be given to bootloaderMonsoon.writeFlash(hex)

verifyHeader(headers):

Verifies the header matches the physical hardware being reflashed.
returns true if the header for the selected fwm file matches the VID and PID of the device connected, false otherwise.

sampleEngine.py

class channels:

Indices used in the sampleEngine class for each of the channels
timeStamp = 0
MainCurrent = 1
USBCurrent = 2
AuxCurrent = 3
MainVoltage = 4
USBVoltage = 5

class triggers:

SAMPLECOUNT_INFINITE = `0xFFFFFFFF`

This value is recognized by the firmware to represent indefinite sampling. The Power Monitor will remain in sample mode until a 'stop' packet is received.

GREATER_THAN:

triggerStyle used in setStopTrigger and setStartTrigger functions. Triggers when measurement > value.

LESS_THAN:

triggerStyle used in setStopTrigger and setStartTrigger functions. Triggers when measurement < value.

class SampleEngine(bulkProcessRate):

bulkProcessRate: the number of samples the engine will collect before it converts a batch of them from raw ADC values into current measurements.

startSampling(samples=5000, granularity = 1):

Puts the Power Monitor into sample mode, and starts collecting measurements.

Samples: the number of samples that will be taken before the device shuts off. Use 0xFFFFFFFF to indicate indefinite sampling.

Granularity: Controls the resolution at which samples are stored. 1 = all samples stored, 10 = 1 out of 10 samples stored, etc. Use larger values if you're having problems with a lot of dropped samples, or if you want smaller file sizes.

setStartTrigger(self,triggerStyle,triggerLevel):

Controls the conditions when the sampleEngine starts recording measurements.
triggerLevel: threshold for trigger start.
triggerStyle: GreaterThan or Lessthan.

setStopTrigger(self,triggerstyle,triggerlevel):

Controls the conditions when the sampleEngine stops recording measurements.

triggerLevel: threshold for trigger start.
triggerStyle: GreaterThan or Lessthan.

setTriggerChannel(self, triggerChannel):

triggerChannel: selected from the channels class. Sets the channel that controls the trigger.

ConsoleOutput(bool):

Enables or disables the display of realtime measurements based on passing a True or False

enableChannel(channel):

Enables a channel. Takes sampleEngine.channel class value as input.

disableChannel(channel):

Disables a channel. Takes sampleEngine.channel class value as input.

enableCSVOutput(filename):

Opens a file and causes the sampleEngine to periodically output samples when taking measurements.

The output CSV file will consist of one row of headers, followed by measurements. If every output channel is enabled, it will have the format:

Time, Main Current, USB Current, Aux Current, Main Voltage, USB Voltage
timestamp 1,main current 1, usb 1, aux 1, main voltage 1, usb 1
timestamp 2, main current 2 . . . Etc.

disableCSVOutput():

Closes the CSV file if open and disables CSV output.

getSamples():

Returns samples in a Python list. Format is [timestamp, main, usb, aux, mainVolts,usbVolts].
Channels that were excluded with the disableChannel() function will have an empty list in their array index.