# Pmonutil Guide

**Author: Ahmed Khawaja**

**Last updated: 5/9/2013**

## Table of Contents

# Overview

The PythonSV pmon utility (referred to as pmonutil) is intended to be the base layer of functionality to pave the way for more advanced solutions. Previously anyone wishing to use performance monitors (pmons) in their scripts requires they access a HAS or spreadsheet to determine which registers they must access, how to program them, event/mask/filter information, etc… This tool is meant to abstract all that away from the user and provide a consistent interface for collecting pmon data. The tool was designed from the ground up to interact heavily with PythonSV, in order to leverage its capabilities and provide a familiar interface to the user. The pmon utility can be interfaced with in many ways which include interactively (command line), inside a script, and potential support for interacting with other tools (like Rocket for instance).

The pmon utility is trending towards incorporating a lot of the features previously found in SVProfiler without some of the limitations that tool experienced. The inputs to the utility are project-specific XML files that contain relevant pmon registers and pmon events. This will allow us to seamlessly switch between projects using the same utility without the huge overhead that SVProfiler experienced in development time. The XML inputs for this utility are the same used by the TAGEC/TACT tools.

The utility is still in development, but it has come a long way. Application Driven Sampling (ADS) is in the works and is a planned feature of the utility.

As with the development of any utility, any feedback that you can provide (bugs and feature requests) would be greatly welcomed. The more people that use this utility, the better the tool will be in the end.

I, Ahmed Khawaja, will cease working on this utility in mid-May. If you have any questions, please contact Lamar Powell (lamar.powell@intel.com) or Amos Christiansen (amos.h.christiansen@intel.com). The last release from Ahmed will be known as v1.0 STABLE.

# Basic Usage

Don't forget to update your PythonSV files before using the utility.

Keep in mind that every function in the utility has doc strings and **help()** can be invoked to figure out what the function does/what parameters it can take.

This part of the guide will concentrate on using the tool interactively. The first thing to getting started is to open up a PythonSV shell (IVT, HSX, HSW currently supported). Once inside a PythonSV shell, we will want to invoke **sv.refresh()**. This is done because in interactive mode, the utility will append pmon attributes to socket objects. Note that this must be done **before** launching the utility. Next, import the utility and launch **main()**. If you are more familiar with pmons and would like to look into the inner workings of the utility, invoke main with a "debug=True" argument.

```
>>> import misc.coverage.pmonutil as pu
>>> pu.main()
DEV BUILD: Version 0.45
Platform: IVT
Stepping: A1
```

After loading, it will dump out some info about the system and utility version.

The utility appends a pmons attribute to each socket under the sv node. This is how the utility is accessed. We select the socket we desire and then access its pmons attribute. Tab completion shows us what is available within the utility.

```
>>> sv.socket0.pmons.
sv.socket0.pmons.cbo              sv.socket0.pmons.imc1             sv.socket0.pmons.r3qpi
sv.socket0.pmons.core             sv.socket0.pmons.irp              sv.socket0.pmons.start
sv.socket0.pmons.globalControls   sv.socket0.pmons.pcu              sv.socket0.pmons.stop
sv.socket0.pmons.ha               sv.socket0.pmons.qpill            sv.socket0.pmons.ubox
sv.socket0.pmons.imc0             sv.socket0.pmons.r2pcie           sv.socket0.pmons.ucrAccessEnabled
```

Most of the things contained at this level of the utility are groups of units and some top level control functions. The globalControls attribute contains freeze/unfreeze controls that broadcast to all pmon counters. This can be used to sync across all pmons. The **start()** and **stop()** functions call the same functions at lower levels. An example is cbo.start calls cbo0.start, cbo1.start, etc...

If we want to program a unit, for instance a cbo, we go into the cbo attribute (then tab complete):

```
>>> sv.socket0.pmons.cbo.
sv.socket0.pmons.cbo.cbo0  sv.socket0.pmons.cbo.cbo13 sv.socket0.pmons.cbo.cbo5  sv.socket0.pmons.cbo.start
sv.socket0.pmons.cbo.cbo1  sv.socket0.pmons.cbo.cbo14 sv.socket0.pmons.cbo.cbo6  sv.socket0.pmons.cbo.stop
sv.socket0.pmons.cbo.cbo10 sv.socket0.pmons.cbo.cbo2  sv.socket0.pmons.cbo.cbo7
sv.socket0.pmons.cbo.cbo11 sv.socket0.pmons.cbo.cbo3  sv.socket0.pmons.cbo.cbo8
sv.socket0.pmons.cbo.cbo12 sv.socket0.pmons.cbo.cbo4  sv.socket0.pmons.cbo.cbo9
```

Here we see all the CBOs on the system. If we desired to program cbo0 pmons, we would go into cbo0 (then tab complete):

```
su.socket0.pmons.cbo.cbo0.clear_all          su.socket0.pmons.cbo.cbo0.getCounterDefault
su.socket0.pmons.cbo.cbo0.clear_counters     su.socket0.pmons.cbo.cbo0.setCounterDefault
su.socket0.pmons.cbo.cbo0.counterStatus      su.socket0.pmons.cbo.cbo0.show
su.socket0.pmons.cbo.cbo0.dump               su.socket0.pmons.cbo.cbo0.start
su.socket0.pmons.cbo.cbo0.events             su.socket0.pmons.cbo.cbo0.stop
su.socket0.pmons.cbo.cbo0.filters
```

Here we are presented with some control functionality, access to filters, and an **events** attribute. This attribute contains the pmon events that can be programmed into this cbo's pmon counters.

```
>>> su.socket0.pmons.cbo.cbo0.events.
su.socket0.pmons.cbo.cbo0.events.cbregs             su.socket0.pmons.cbo.cbo0.events.ring_sink_starved
su.socket0.pmons.cbo.cbo0.events.clockticks         su.socket0.pmons.cbo.cbo0.events.ring_src_thrtl
su.socket0.pmons.cbo.cbo0.events.core_snp           su.socket0.pmons.cbo.cbo0.events.rxr_debug
su.socket0.pmons.cbo.cbo0.events.counter0_occupancy su.socket0.pmons.cbo.cbo0.events.rxr_ext_starved
su.socket0.pmons.cbo.cbo0.events.debug_ismq         su.socket0.pmons.cbo.cbo0.events.rxr_inserts
su.socket0.pmons.cbo.cbo0.events.debug_pipe         su.socket0.pmons.cbo.cbo0.events.rxr_int_starved
su.socket0.pmons.cbo.cbo0.events.fast_asserted      su.socket0.pmons.cbo.cbo0.events.rxr_ipq_retry
su.socket0.pmons.cbo.cbo0.events.ismq_drd_miss_occ  su.socket0.pmons.cbo.cbo0.events.rxr_irq_retry
su.socket0.pmons.cbo.cbo0.events.llc_victims        su.socket0.pmons.cbo.cbo0.events.rxr_ismq_retry
su.socket0.pmons.cbo.cbo0.events.misc               su.socket0.pmons.cbo.cbo0.events.rxr_occupancy
su.socket0.pmons.cbo.cbo0.events.nameless           su.socket0.pmons.cbo.cbo0.events.tor_inserts
su.socket0.pmons.cbo.cbo0.events.qlru               su.socket0.pmons.cbo.cbo0.events.tor_occupancy
su.socket0.pmons.cbo.cbo0.events.qlru_1             su.socket0.pmons.cbo.cbo0.events.txr_ads_used
su.socket0.pmons.cbo.cbo0.events.ring_ad_used       su.socket0.pmons.cbo.cbo0.events.txr_debug
su.socket0.pmons.cbo.cbo0.events.ring_ak_used       su.socket0.pmons.cbo.cbo0.events.txr_inserts
su.socket0.pmons.cbo.cbo0.events.ring_bl_used       su.socket0.pmons.cbo.cbo0.events.txr_occupancy
su.socket0.pmons.cbo.cbo0.events.ring_bounces       su.socket0.pmons.cbo.cbo0.events.txr_starved
su.socket0.pmons.cbo.cbo0.events.ring_iv_used       su.socket0.pmons.cbo.cbo0.events.xsnp_resp_any_rspi
```

Here we can see that inside a cbo, we can monitor many events. Some events listed here have no submasks, which means the event has no sub-events. Some events can have submasks, which means you can choose a sub-events of that event or combinations of sub-events. We examine both instances below.

First we want to setup cbo0 to measure the "clockticks" event. We select clockticks and inspect what functions we have available to us:

```
>>> su.socket0.pmons.cbo.cbo0.events.clockticks.
su.socket0.pmons.cbo.cbo0.events.clockticks.getspec su.socket0.pmons.cbo.cbo0.events.clockticks.setup
```

We see that we have **getspec** and **setup** functions here. Invoking **getspec()** will dump information out about the event.

```
>>> su.socket0.pmons.cbo.cbo0.events.clockticks.getspec

========== Event/Mask Info ========

Name: clockticks
Event Code: 0x0
Description: Uncore Clocks
Mask: 0x0
Type: singlepmon
Cluster: cbo
Counters: [0, 1, 2, 3]
Filter: F
Inverted: 0
Internal: True
Threshold: 0
Read Only: False
Edge Detect: 0
Fixed: False
Default: False
Inc per cycle: 1
MSR: None
```

Invoking this function has shown us what will be programmed into the pmon control registers.

Below are explanations of what some of this data means.

**Event Code**: The code programmed into the pmon control register to select this event.

**Mask**: The sub-event code.

**Counters**: Which pmon counters in this unit that are capable of measuring this event. The utility does this automatically for you.

**Filter**: Is this event affected by filter registers in this unit (example like threadid or opcode)

**Internal**: True here means the event is only measurable on an unlocked system.

**Threshold**: This is relevant if an event is measuring queue occupancy for example. If threshold is 8, the event only increments if the queue has 8 or more entries in it.

**Inverted**: Changes the inequality used with threshold from >= to < .

**Fixed**: Does this event use a fixed pmon counter, the utility also abstracts this away from the user.

**Edge Detect**: Does the event only increment based on a change of state as opposed to on a per cycle basis.

**Inc per cycle**: How much the counter is incremented by on each cycle.

The next thing we wish to do is setup a pmon to measure this event, which we do by invoking **.setup()**:

```
>>> su.socket0.pmons.cbo.cbo0.events.clockticks.setup()
```

The utility takes care of selecting the counters and doing all the small details. Exceptions will be thrown however if no counters are available or certain types of counters are unavailable.

The next thing we wish to do is program an event with a sub-event mask, we go up one level and select the *ring_ad_used* cbo event and tab complete:

```
>>> su.socket0.pmons.cbo.cbo0.events.ring_ad_used.
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.down_vr0_even_0x4
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.down_vr0_odd_0x8
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.down_vr1_even_0x40
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.down_vr1_odd_0x80
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.help
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.setupCombo
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.showCombos
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.up_vr0_even_0x1
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.up_vr0_odd_0x2
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.up_vr1_even_0x10
su.socket0.pmons.cbo.cbo0.events.ring_ad_used.up_vr1_odd_0x20
```

All the attributes with hex numbers at the end are sub-events. The hex numbers represent the sub-event mask. If we wanted to setup *down_vr0_even_0x4* and *up_vr1_odd_0x20* (which is done by ORing the sub-masks together) we would do it by invoking **setupCombo** (note that sub-masks are just called masks in the function call), demonstrated below:

```
>>> su.socket0.pmons.cbo.cbo0.events.ring_ad_used.setupCombo(masks=[0x4,0x20])
```

You may also just go a level deeper for one sub-mask and invoke **.setup()**. Please note that **.setup()** returns a eventCounterHandle object, refer to the appropriate section in this guide for more details.

Once we have some events setup, we go back to the cbo0 level, and invoke **.show()** to display some information about how this unit's pmons are currently configured:

```
>>> su.socket0.pmons.cbo.cbo0.show()
Counter 0 Config: Setup, event=clockticks, subevents=None
Counter 0 Count: 0x0L
Counter 1 Config: Available, event=None, subevents=None
Counter 1 Count: 0x0L
Counter 2 Config: Setup, event=ring_ad_used, subevents=down_vr0_even_0x4,up_vr1_odd_0x20
Counter 2 Count: 0x0L
Counter 3 Config: Available, event=None, subevents=None
Counter 3 Count: 0x0L
```

Here we can see that the events we setup have been assigned to counters and that the counters are setup but not started. Now we can invoke **.start()** at this level and then .**show()** to view the contents of the counters. Similarly, **.stop()** can be called to freeze the counters.

The states that counters can be in are:

- **Available**: Ready to be programmed to measure an event.
- **Setup**: Control information programmed in, but counter is not yet measuring events.
- **Measuring**: Counter is actively measuring an event.
- **Stopped**: Counter was setup, measured some events, and was then stopped by the user. It still retains the information needed to monitor an event and can be simply restarted.

The utility also has clear functions, access to unit filters, and **.start()** and **.stop()** from many levels (for instance cbo0 vs. all cbos). The clear functions can clear config and counter registers or just the counters themselves.

Filters are additional pmon registers associated with a unit that can further describe the event data you wish to collect. Some filters for instance check for opcodes or threadids. Filters currently have to be manually programmed and can be accessed via a unit.filters attribute (example cbo.cbo0.filters.getFilters() ).

As stated earlier**, help()** can be called on any function in the utility and it will explain how to use the function and what it does.

Things to be aware of:

- Units have limited amounts of counters (at most 5).
- Not all counters are equal.
    - Some counters can count all events, and some can only count a subset.
- Not all events have been tested, so a non-incrementing counter might be indicative of hardware or software issues.
- Make sure some traffic is on the system, and the system isn't halted!

# Programming and controlling counters, Event Counter Handles

The main goal of the pmon utility is to enable easy programming of pmon counters and easy retrieval of the values from those counters. The process of programming in an event is documented in the introduction section to this guide. When trying to program in an event, the utility will try to find an available counter and will throw an exception if it can't. After executing **.setup()** on an event, an eventCounterHandle object is returned. This object can be used to control a specific counter/event combination if the normal unit level controls are not fine grain enough. The functions/members available in this class are:

**clearConfig()** – This empties the configuration register and marks the evenCounterHandle as no longer valid.

**clearCounter()** – This clears the value of the counter, but not the configuration register.

**dumpCSV()** – Dumps CSV information string used to output to a CSV file.

**dumpInfo()** – Returns a dictionary of information about the counter and event.

**getValue()** – Does a read on the counter and returns the value.

**isMeasuring()** – Returns a Boolean stating if the pmon counter is counting or not.

**start()** – Start measuring.

**stop()** – Stop measuring.

**validHandle** – Determines if the handle is valid or should be disregarded.

Keep in mind that when the event information is cleared from the configuration register, this object will become invalid because the event to counter mapping it once represented is no longer valid.

Using this object is preferred for reading values and using the unit level control is preferred for starting/stopping all the registers at the same time. Keep in mind that under the hood, this object is nothing more than a wrapper for the unit controls with arguments to specify the counters.

# Command Line Arguments and Starting the Utility

After importing the utility, **.main()** should be invoked to setup everything and prepare counters to be programmed. When **.main()** is invoked, the register and event XMLs are parsed in and appropriate objects are added under each PythonSV socket for programming and controlling counters with various events. Thus, **.main()** must be invoked before the utility can be used, this is true both in command line usage and in a script.If invoking the utility in the command line, **sv.refresh()** must be invoked before **pu.main()** is invoked, this is not the case for scripts. When starting the utility, there are many command line arguments that can be passed to **.main()** to augment the behavior of the utility. Below is a table describing the key-worded arguments that **.main()** accepts.

| Keyword | Value | Description |
|---|---|---|
| debug | True/False (defaults to False) | Prints really detailed log information to the screen when set to True |
| logfilename | Desired name for log file, defaults to "pmonutil_last_run_log.txt" | The utility generates a log file every run, give a unique name to make sure the log doesn't get overwritten |
| log_overwrite | True/False (defaults to True) | Determine whether to append |
| events_xml | Path to file (default depends on the project) | XML file containing pmon events information |
| default_events_xml | Path to file (default depends on the project) | XML file containing pmon fixed events |
| registers_xml | Path to file (default depends on the project) | XML file containing information on pmon registers |
| control_xml | Path to file (default depends on the project) | XML file containing pmon control registers, only used in IVT |
| initialize_all | True if you want all units to be initialized(defaults False) | Determines if the registers should be found during startup |
| initialize_units | A list of strings containing which specific units you want to initialize | ['cbo','sbo'] would initialize all cbo and sbo units |
| searchMode | "address" or "name" (defaults to "name", which is faster) | How pmon registers should be found in PythonSV |
| counterMode | True/False (defaults to False) | Setting this to True stops implicit clears and disables unit level control, only per counter control would be enabled. |

**Keep in mind that main() can be invoked with no arguments, and for most users that is how it should be run.**

The **counterMode** option was put in place to allow multiple instances of the utility to run at the same time. The only problem with this is since the utility was designed for single instance use originally, you lose all the automatic handling of counters and the user must in their code enforce that multiple instances are not using the same counters. This also stops implicit clears from occurring and the user must manually clear pmon registers. This is a very niche feature and only those well versed in how the utility works should use it if their work so requires.

The **debug** option prints out register values before and after writes and displays a lot of useful information about what the utility is doing under the hood. This is a good thing to enable if you want to understand how the pmons are being programmed and if you want to double check that the utility is doing what it is supposed to.

The other options should usually be left to their default values.

# Code Location/Importing the utility

All the code for the pmon utility can be found in the pythonsv/coverage/misc directory.

The utility relies on XMLs that contain register and event programming information. For every project, the XMLs can be found under the coverage directory for that project. Example, HSX XMLS are found in pythonsv/haswellx/coverage.

To import the utility into either a script or the command line, simply call:

import misc.coverage.pmonutil as pu

Importing it under an alias makes it easier to invoke the utility and for the rest of this document, pu will be a reference to misc.coverage.pmonutil.

It is also important to note that the utility is not yet active and must be invoked(calling pu.main) before being usable, as will be demonstrated below.

Below is a table that briefly explains what each file in the utility contains.

| File | Description |
|---|---|
| pmonutil.py | Majority of the code for doing most things |
| libpu.py | Contains some helper functions used throughout the utility |
| pmon_reg_parser.py | Code for parsing the register XML files |
| events_parser.py | Code for parsing the events XML files |
| util_exceptions.py | Contains exceptions that the utility can throw |
| tor_mm.py | Code for the TOR Mask/Match plugin |

# Possible Enhancments

- Application driven sampling (ADS) on SVOS
- Time based sampling (wall time and clock cycles)
- VT-d pmons
- PCIe pmons
- Smarter unit auto discovery
- Built in equations for data post-processing
- Ability to easily add your own post-processing equations
- Histogram functionality
    - Measuring multiple thresholds from a min value to a max value
- Possible integration with other coverage tools