Sapphire Rapids Uncore Programming Guide

Reference Manual

June 2021

Revision 1.0

Reference Number: 642245

Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Xeon are trademarks of Intel Corporation or its subsidiaries.

*Other names and brands may be claimed as the property of others.

Contents

Figures

Tables

Revision History

Revision Number	Description	Date
1.0	• Initial Release	June 2021

1 Introduction

"Uncore" roughly equates to logic outside the CPU cores but residing on the same die. Traffic (for example, data reads) generated by threads executing on CPU cores or I/O devices may be operated on by logic in the uncore. The logic is responsible for managing coherency, managing access to the DIMMs, managing power distribution and sleep states, and so on.

The uncore sub-system of the next generation Sapphire Rapids server is shown in Figure 1-1. The uncore sub-system consists of a variety of components, many assigned to the aforementioned responsibilities, ranging from the Caching/Home Agent (CHA) to the Power Control Unit (PCU) and IMC, to name a few. Most of these components provide similar performance monitoring capabilities.

Before going into the details of Sapphire Rapids server's uncore PMON, the following sections provide:

- A general overview of the uncore PMON operation and the state provided SW to manage its operation.
- Functionality common to individual units with the common logic to support the functionality.
- A summary of Sapphire Rapids server's uncore PMON capabilities.
- An overview of all Sapphire Rapids server uncore PMON states.
- An introduction to a new discovery mechanism.
- Guidance to the SW, including how to manage a monitoring session, find the base address to the page of discovery, and find the base addresses for the PMON registers addressed in PCICFG or Memory-Mapped Input/Output (MMIO) space.

1.1 Section References

The following sections provide a breakdown of the performance monitoring capabilities for each box:

- Section 2.1, Mesh Performance Monitoring
- Section 2.2, CHA Performance Monitoring
- Section 2.3, IMC Performance Monitoring
- Section 2.4, IIO Performance Monitoring
- Section 2.5, IRP Performance Monitoring
- Section 2.6, Intel® UPI Link Layer Performance Monitoring
- Section 2.7, M2M Performance Monitoring
- Section 2.8, M2PCIe* Performance Monitoring
- Section 2.9, M3 Intel® UPI Performance Monitoring
- Section 2.11, PCU Performance Monitoring
- Section 2.11, MDF Performance Monitoring

1.2 Uncore PMON Overview

1.2.1 A Simple Hierarchy

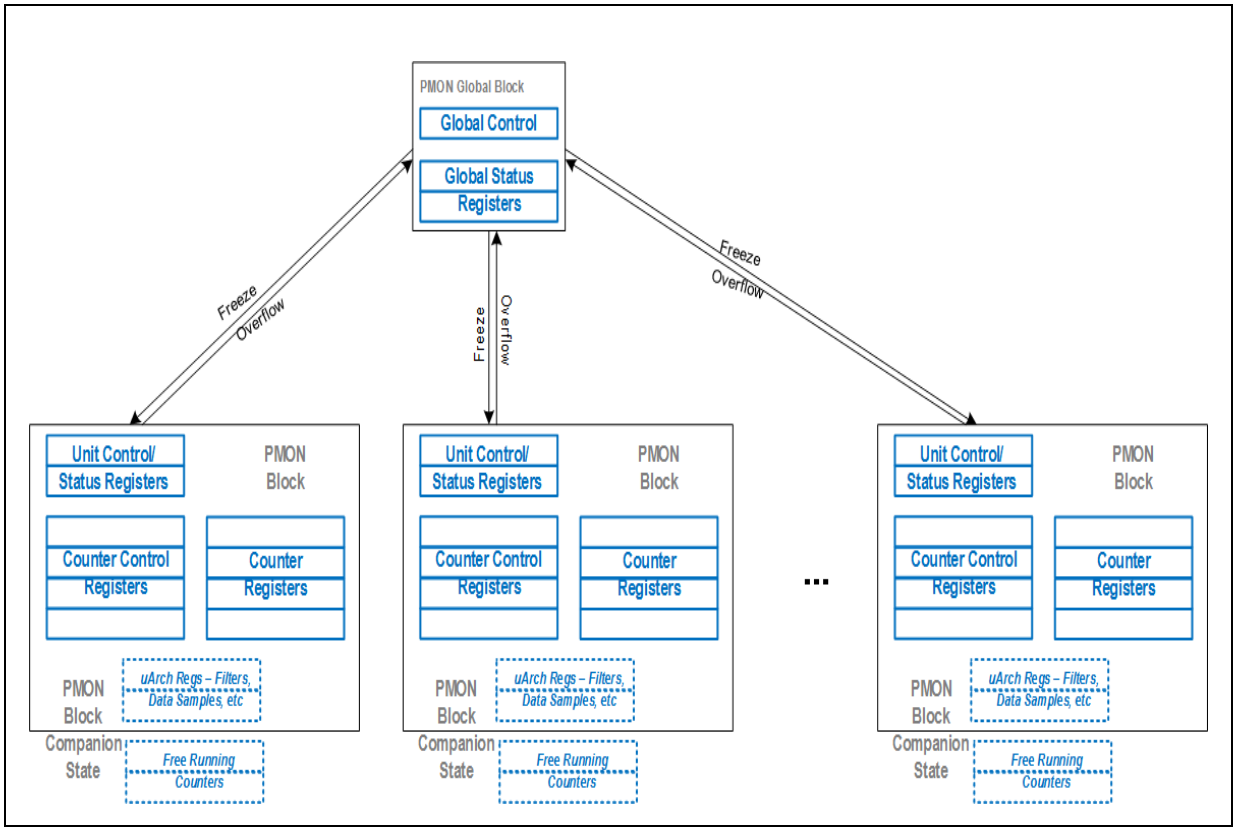
The uncore PMON is managed through a very simple hierarchy. There are some number of PMON units governed by a global control.

Each PMON block contains a set of counters (Ctrs) with paired control registers. Each unit provides a set of events for the SW to select from. The SW can ask the HW to collect an event by specifying what to count in a counter's control register. Then, the SW can periodically read the collected value from the paired counter.

Some units offer an expanded event set that require additional counter control bits. (for example, CHA, IIO and Intel® Ultra Path Interconnect [Intel® UPI]).

Some units offer the ability to further refine, or "filter", the monitored events through additional counter control registers.

Figure 1-1. Uncore PMON Components and Hierarchy



Note: The uncore PMONs represent a per-socket resource not meant to be affected by context switches and thread migration performed by the OS. It is recommended that the monitoring software agent establishes a fixed affinity binding to prevent event count cross-talk across the uncore PMON collected from different sockets.

To manage the large number of counter registers distributed across so many units and collect event data efficiently, each block has a modest amount of control and status governed by a similar global control and status.

The SW can directly synchronize actions across counters (for example, to start, stop, and reset counting) within each PMON block or across all PMON blocks through this control state.

The SW can indirectly synchronize actions across counters (for example, stop counting) in all the PMON blocks by telling the HW what to do when a counter overflows. After a set number of events have been captured by pre-seeding the counter, the SW can set a counter to overflow. For each counter, the SW can then choose whether to notify the global PMON control that a counter has overflowed.

Upon receipt of an overflow, the global control will assert the global freeze signal. Once the global freeze has been detected, each box will disable (or "freeze") all of its counters. In the process of generating a global freeze, the SW can configure the global control to send a Performance Monitoring Interface (PMI) signal to the core executing the monitoring software.

The following sections detail the basic control state provided to the SW to control performance monitoring in the uncore.

1.2.2 Global PMON State

1.2.2.1 Global PMON, Global Control, and Status Registers

The following registers represent the state governing all Performance Monitor Units (PMUs) in the uncore, both to exert global control and collect unit-level information.

U_MSR_PMON_GLOBAL_CTL contains bits that can stop *(.frz_all)* all the uncore counters.

Figure 1-2. PMON Global Control Register for Sapphire Rapids Server

Table 1-1. U_MSR_PMON_GLOBAL_CTL Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**60:1**	**RV**	**0**	**Reserved**
frz_all	0	WO	0	Freeze all uncore PMONs

If an overflow is detected in any of the uncore's PMON registers, it will be summarized in one or more U_MSR_PMON_GLOBAL_STATUS registers. These registers accumulate overflows sent to it from the uncore boxes with PMON blocks. To reset these overflow bits, a user must set the corresponding bits in U_MSR_PMON_GLOBAL_STATUS to 1, which will act to clear them.

Figure 1-3. PMON Global Status Register for Sapphire Rapids Server

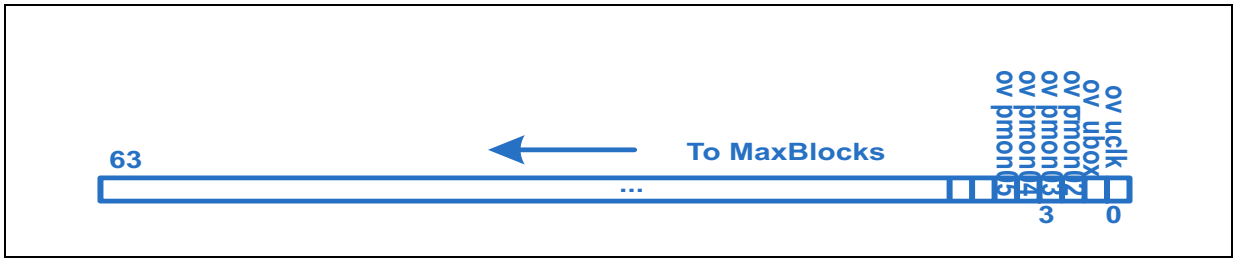


Table 1-2. U_MSR_PMON_GLOBAL_STATUS Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**63:MaxBlocks**	**RV**	**0**	**Reserved**
ov_pmonX	MaxBlocks-1:4	RW1C	0	Overflow detected in PMON register from Block with "Global Status Position" of "MaxBlocks-1" as reported through global discovery
ov_pmonx-1: ov_pmon04	MaxBlocks-2:4	RW1C	0	Overflow detected in PMON registers from the blocks with a "Global Status Position" between MaxBlocks-1 and 3
ov_pmon03	3	RW1C	0	Overflow detected in PMON register from block with "Global Status Position" of 3
ov_pmon02	2	RW1C	0	Overflow detected in PMON register from block with "Global Status Position" of 2
ov_u	1	RW1C	0	Overflow detected in the UBox PMON register
ov_uclk	0	RW1C	0	Overflow detected in the UBox fixed Unified Memory Controller Clock (UCLK) register

The mapping of global status bits in the global status registers to PMON blocks will be provided through the new PMON discovery mechanism. The first two status bits correspond to the UCLK fixed register and the UBlock respectively. The rest of the status bits correspond to overflows detected from the PMON Block's identified through discovery. The discovery for each PMON block will report its "Global Status Position" (for example, which bit in the global status register records its overflows).

For instance, the SW may discover a PMON block of unit type equal to CHA, the unit ID 5 has a "Global Status Position" of 5.

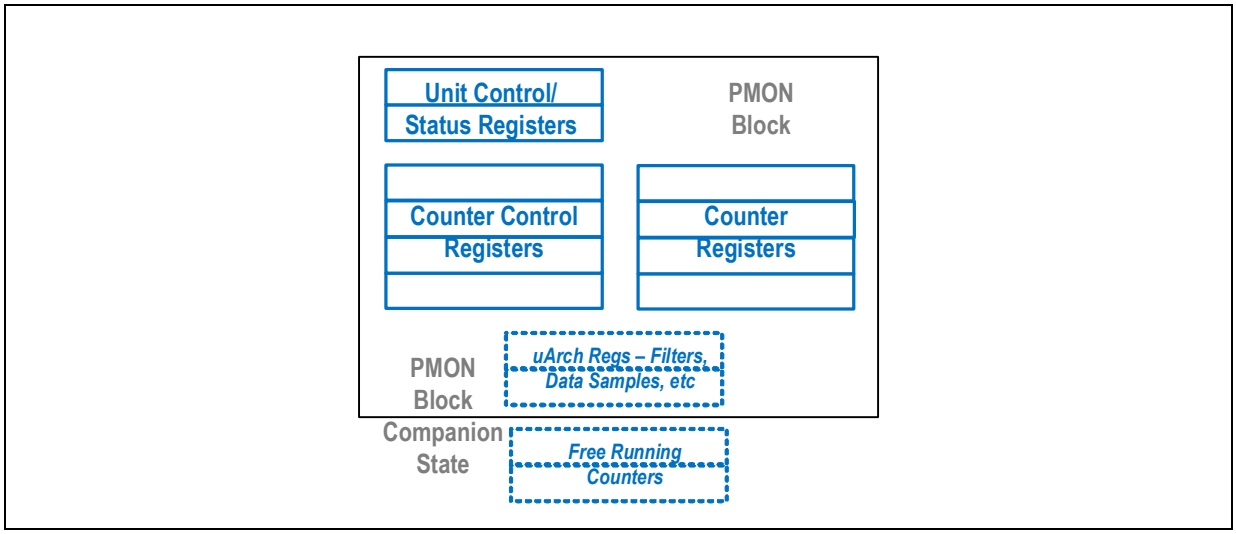
1.3 Unit Level PMON State

Each PMON block in the uncore is composed of the following state:

- A unit control register to aid the SW sample collection
- Status registers to record when a counter within the block overflows
- A set of data registers
- A set of control registers, each paired to a data register, to allow the SW to specify what event should be captured

- Additional micro-architectural specific state designed to enhance performance monitoring collection within a block (for example, event or traffic filters).
- Some free running counters, although not subject to the PMON hierarchy, is included in this document with the unit they are associated with.

Figure 1-4. PMON Blocks



Every PMON block in the system is governed by a modest amount of unit level control. Each bit intended to assist the SW in more efficiently managing the PMON state within the block. Reset bits help reduce the time the SW needs to setup a new sample.

Note: If the PMON registers within the unit are shared among different users, either those users can leave this register untouched or they can agree on the user allowed to affect the unit level control state.

Figure 1-5. PMON Unit Control Register for Sapphire Rapids Server - Common to All PMON Blocks

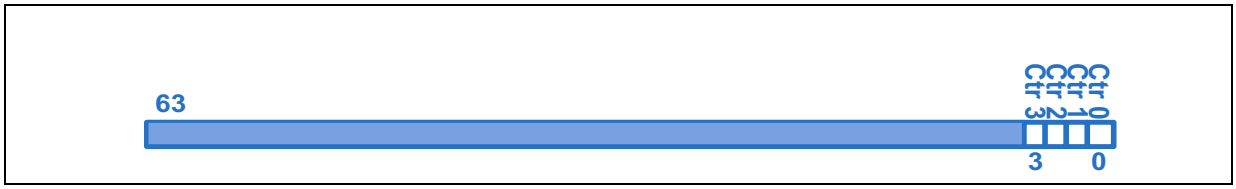
Table 1-3. PMON_UNIT_CTL Register – Field Definitions (Sheet 1 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
rst_ctrs	9	WO	0	Reset counters When set to 1, the counter registers will be reset to 0

Table 1-3. PMON_UNIT_CTL Register – Field Definitions (Sheet 2 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
rst_ctrl	8	WO	0	Reset control When set to 1, the counter control registers will be reset to 0
rsv	**7:1**	**RV**	**0**	**Reserved**
frz	0	WO	0	Freeze If set to 1, the counters in this box will be frozen

Figure 1-6. PMON Unit Status Register for Sapphire Rapids Server - Format Common to All PMON Blocks



If an overflow is detected from one of the unit's PMON registers, the corresponding bit in the PMON_UNIT_STATUS.ov field will be set. To reset these overflow bits, a user must write a value of "1" to them (which will clear the bits). There are typically four counters per PMON block. But that number may vary. As of Sapphire Rapids, the number of paired counter and counter-control registers is reported through the unit discovery associated with each PMON block. The unit status register will contain "NumControlRegs" valid bits.

Note: Check Table 1-7 or the section detailing each unit's functionality for the number counters it supports.

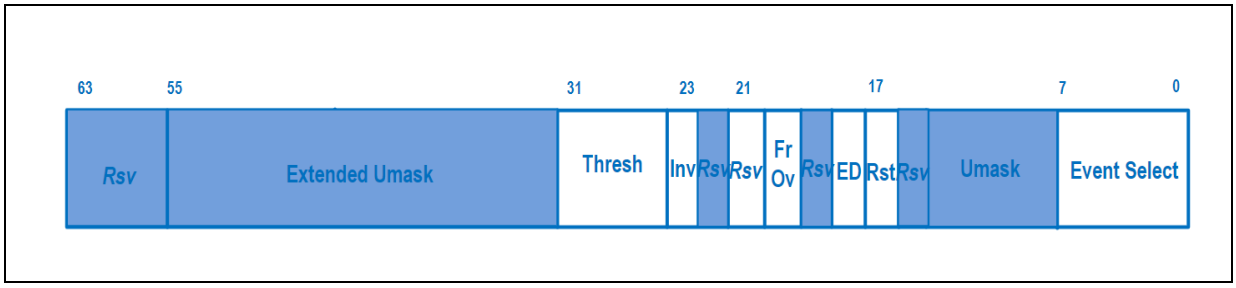
Table 1-4. PMON_UNIT_STATUS Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**31:4**	**RV**	**0**	**Reserved**
ov	NumControlRegs-1:0	RW1C	0	If an overflow is detected from the corresponding PMON_CTR register, its overflow bit will be set ***Note:*** Write of "1" will clear the bit Although 4 is very common, the number of overflow bits can vary by the PMON block. The number can be discovered in the NumControlRegs field of the unit's discovery

1.3.0.1 Unit PMON state - Counter and Control Pairs

The following table defines the layout for the standard performance monitor control registers. Their main task is to select the event to be monitored by their respective data counter (.ev_sel, .umask). Additional control bits are provided to shape the incoming events (for example, .invert, .edge_det, and .thresh) as well as provide additional functionality for monitoring the software (.rst,.ov_en).

Figure 1-7. PMON Counter Control Register for Sapphire Rapids Server - Fields Common to All PMON Blocks



Notes: Per unit considerations - See each unit section for more details on:

- Certain units may make use of additional bits in these counter control registers.
- The width of the thresh field is dependent on a unit's "widest" event (for example, the event that can increment the most per cycle, typically measuring per-cycle occupancy of a large queue).
- Several unit counter control registers are still 32b, some 64b. All are addressable as 64b registers.

The next section shows an overview of the counter control logic.

Table 1-5. Baseline *_PMON_CTLx Register – Field Definitions (Sheet 1 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**63:56**	**RV**	**0**	**Reserved** **Only relevant to units that use 64b control registers.**
Extended umask	55:32	RW	0	Extension to umask. Adds additional filtering capabilities to certain special events.
Thresh	31:24	RW	0	Threshold is used, along with the invert bit, to compare against the counter's incoming increment value. For example, the value that will be added to the counter. For events that increment by more than 1 per cycle, if the threshold is set to a value greater than 1, the data register will accumulate instances in which the event increment is greater or equal to the threshold. For example, if there is an event to accumulate the occupancy of a 64-entry queue every cycle; by setting the threshold value to 60, the data register would count the number of cycles the queue's occupancy was greater or equal to 60.

Table 1-5. Baseline *_PMON_CTLx Register – Field Definitions (Sheet 2 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
invert	23	RW	0	Invert comparison against the threshold. 0 - comparison will be "is the event increment greater or equal to the threshold?". 1 - comparison is inverted - "is the event increment less than the threshold?" For example, for a 64-entry queue, if the SW wanted to know how many cycles the queue had fewer than 4 entries, the SW has to set the threshold to 4 and set the invert bit to 1. ***Note:*** Invert is in series following .thresh, due to this, the .thresh field must be set to a non-0 value. For events that increment by no more than 1 per cycle, set the .thresh to 0x1. Also, if the .edge_det is set to 1, the counter will increment when a 1 to 0 transition (for example, falling edge) is detected.
rsv	**22:21**	**RV**	**0**	**Reserved** **The SW must write to 0 else behavior is undefined.**
ov_en	20	RW	0	When this bit is set to 1 and the corresponding counter overflows, an overflow message is sent to the UBox's global logic. The message identifies the unit that sent it. Once received, the global status register will record the overflow in the corresponding U_MSR_PMON_GLOBAL_STATUS bit.
rsv	**19**	**RV**	**0**	**Reserved**
edge_det	18	RW	0	When set to 1, rather than measuring the event in each cycle that it is active, the corresponding counter will increment when a 0 to 1 transition (for example, rising edge) is detected. When 0, the counter will increment in each cycle that the event is asserted. ***Note:*** .edge_det is in series following the .thresh, due to this, the .thresh field must be set to a non-0 value. For events that increment by no more than 1 per cycle, set the .thresh to 0x1.
rst	17	WO	0	When set to 1, the corresponding counter will be cleared to 0.
rsv	**16**	**RV**	**0**	**Reserved** **The SW must write to 0, else the behavior is undefined.**
umask	15:8	RW	0	Select subevents to be counted within the selected event.
ev_sel	7:0	RW	0	Select event to be counted.

The default width for performance monitor data registers are 48b wide. A counter overflow occurs when a carry out from bit 47 is detected. The SW can force all uncore counting to freeze after N events by preloading a monitor with a count value of 248 - N and setting the control register to send an overflow message to the UBox, see Section 1.2.2. During the interval of time between the overflow and global disable, the counter value will wrap and continue to collect events.

To ensure accuracy, the SW has to stop the counter and check the overflow status before reading its value. But, if accessible, the SW can continuously read the data registers without disabling event collection.

Figure 1-8. PMON Counter Register for Sapphire Rapids Server - Common to All PMON Blocks

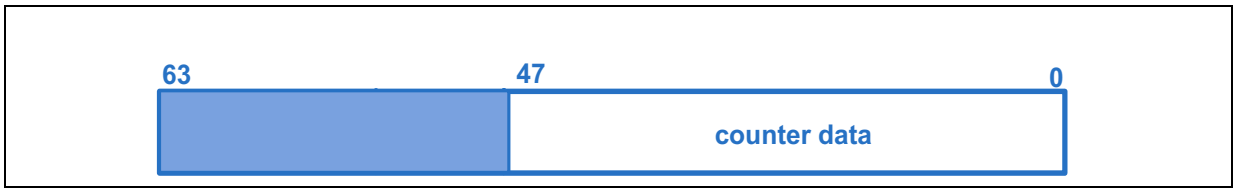


Table 1-6. Baseline *_PMON_CTRx Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**63:48**	**RV**	**0**	**Reserved**
event_count	47:0	RW-V	0	48-bit performance event counter

1.3.0.2 Unit PMON Registers - On Overflow and the Consequences (PMI and Freeze)

If an overflow is detected from a unit's performance counter, the overflow bit is set at the unit level (*_PMON_UNIT_STATUS.ov).

If the counter is enabled to communicate the overflow (*_PMON_CTL.ov_en is set to 1), an overflow message is sent to the UBox. When the UBox receives the overflow signal, the *_PMON_GLOBAL_STATUS.ov_x bit is set, a global freeze signal is sent and a PMI can be generated.

Note: The "x" represents the box generating the overflow, see Table 1-3.

Once a freeze has occurred, in order to see a new freeze, the overflow responsible for the freeze must be cleared by setting the corresponding bit in the *_PMON_UNIT_STATUS.ov and the U_MSR_PMON_GLOBAL_STATUs.ov_x to 1 (which acts to clear the bits).

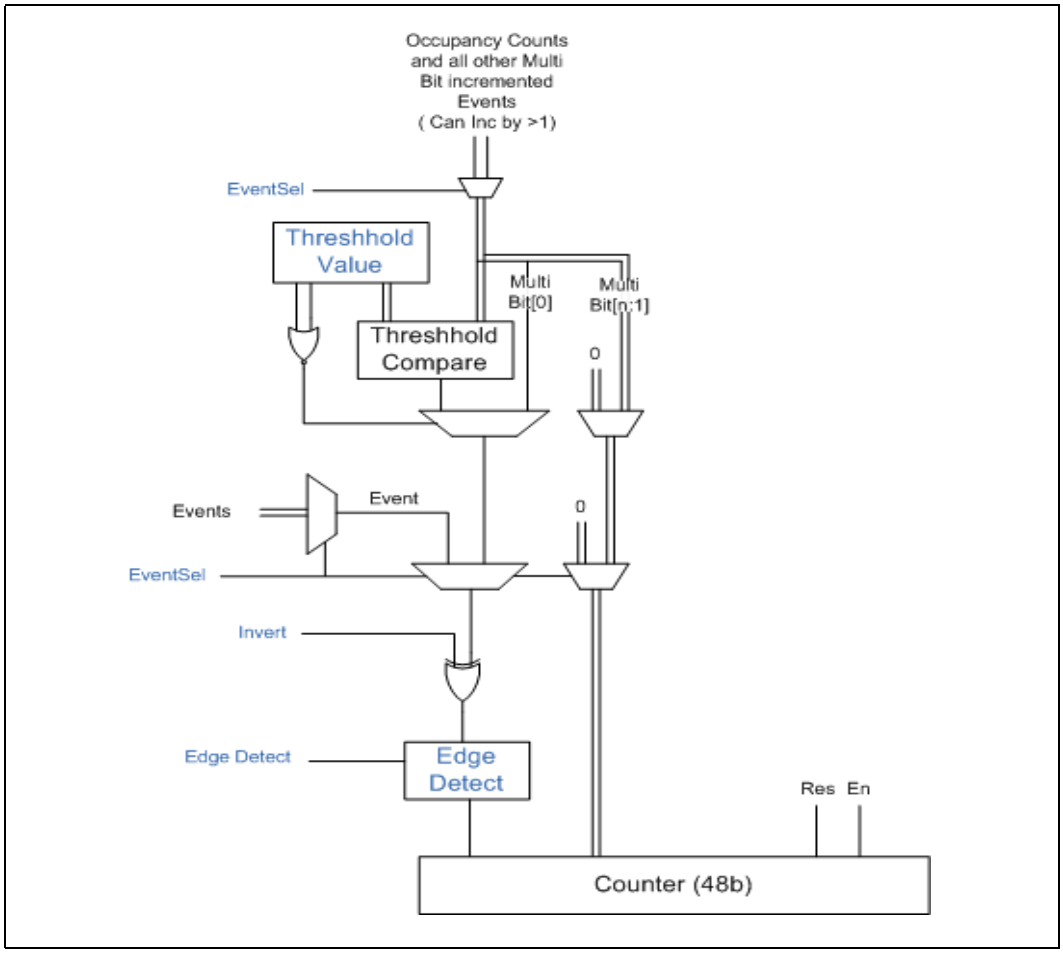
Assuming all counters have been locally enabled (the .en bit is set to 1 in every control register meant to monitor events) and the overflow bits have been cleared, the unit is prepared for a new sample interval. Once the global controls have been re-enabled counting will resume, see Section 1.10.6.

1.4 Uncore PMON - Typical Counter Control Logic

Following is a logic diagram for the standard Performance Monitor* (PerfMon*) counter control. It illustrates how the event information is routed, selected, filtered (by other bits in the control register), and sent to the paired data register for storage.

Note: The PCU uses an adaptation of this block, see Section 2.10.1 for more information. Also note that only a subset of the available control bits is presented in the diagram.

Figure 1-9. PerfMon* Counter Control Block Diagram



- **Selecting what to monitor:** The main task of a configuration register is to select the event to be monitored by its respective data counter. Setting the *.ev_sel* and *.umask* fields performs the event selection.

Note: Only the *.ev_sel* is pictured in the previous figure. The *.umask* field is generally used to select subevents of the event. Once the proper subevent combination has been selected, it is passed on to the per counter EventSel Multiplexer (MUX).

Additional control bits used to filter and create information related to the selected Event:

- **Applying a threshold to incoming events:** *.thresh* - Since most counters can increment by a value greater than 1, a threshold can be applied to generate an event based on the outcome of the comparison. If the *.thresh* is set to a non-zero value, that value is compared against the incoming count for that event in each cycle. If the incoming count is greater or equal than the threshold value, then the event count captured in the data register will be incremented by 1.

  Using the threshold field to generate additional events can be particularly useful when applied to a queue occupancy count. For example, if a queue is known to

contain eight entries, it may be useful to know how often it contains six or more entries (such as almost full) or when it contains 1 or more entries (for example, "Not Empty").

For Sapphire Rapids the *.invert* and the *.edge_det* bits follow the threshold comparison in the sequence. If a user wishes to apply these bits to events that only increment by 1 per cycle, the thresh must be set to 0x1.

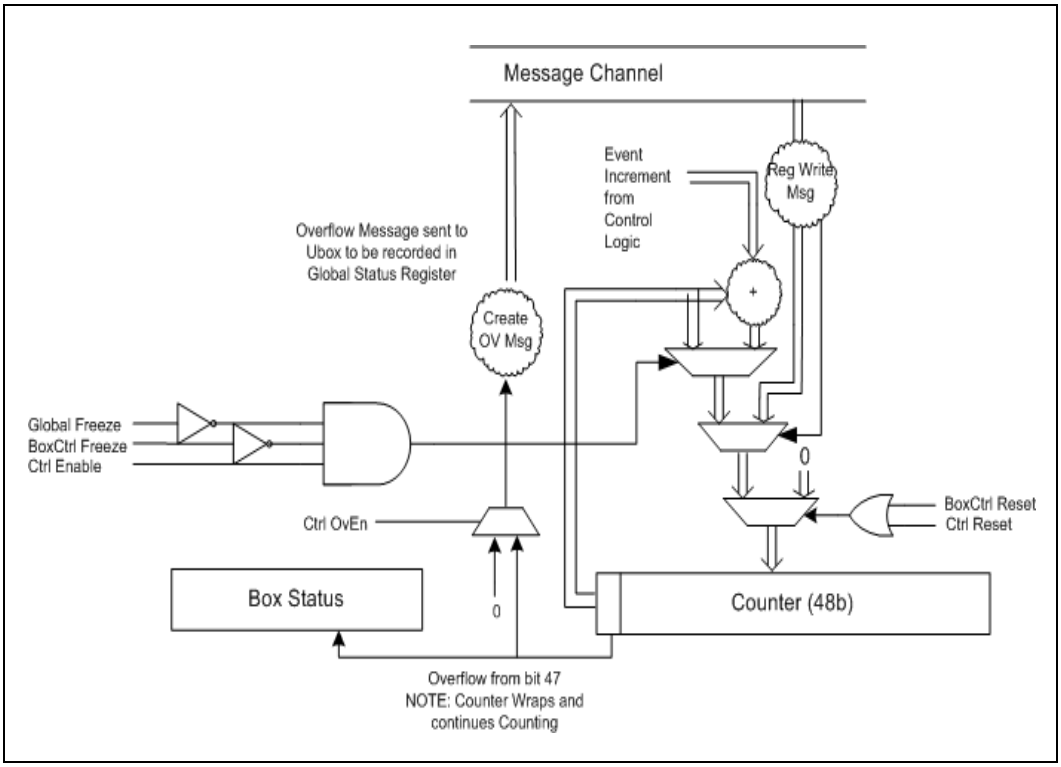
- **Inverting the threshold comparison:** *.invert* - Changes the *.thresh* test condition to "<".
- **Counting state transitions instead of per-cycle events:** *.edge_det* - Rather than accumulating the raw count each cycle (for events that can increment by 1 per cycle), the register can capture transitions from no event to an event incoming (for example the "Rising Edge").

1.5 Uncore PMON - Typical Counter Logic

The following is a diagram of a standard PerfMon* counter. It illustrates the control involved in managing the data register. The functionality includes how to start or stop the register, reset it, indicate an overflow and capture the information sent from the counter control block. The diagram contains bits from all levels in the counter control hierarchy global, unit level as well as from the paired counter control register.

Details on how to perform counter management, including how to set up a monitoring session and periodically sample the counters can be found in Section 1.10.4.

Figure 1-10. PerfMon* Counter Block Diagram



- **Telling the HW that the control register is set:** the *.en* bit must be set to 1 to enable counting. Once the counting has been enabled at all levels of the performance monitoring hierarchy, the paired data register will begin to collect events, see Section 1.10.4 for more information.
- **Notification after X events:** the *.ov_en* - Instead of manually stopping the counters at intervals (often wall clock time) pre-determined by the software, the hardware can be set to notify monitoring software when a set number of events has occurred. The overflow enable bit is provided for just that purpose. See Section 1.3.0.2 for more information on how to use this mechanism.

1.6 Sapphire Rapids Server Uncore PMON

The general performance monitoring capabilities of each box are outlined in the following table.

Table 1-7. Per-Box Performance Monitoring Capabilities

Box	Numbers Counters/ Box	Packet Match or Mask Filters?	Bit Widths
CHA	4	Y	48
IIO	4 (+1) per stack (+4 per port)	N	48
IRP	2	N	48
IMC	4	N	48
Intel® UPI	*4 (per link)*	Y	48
M3 Intel® UPI	*4 (per link)*	N	48
M2M	*4*	Y	48
M2PCIe*	*4*	N	48
PCU	4 (+2)	N	48
MDF	*4*	N	48

The programming interface of the counter registers and control registers fall into three address spaces:

- CHA, M2PCIe*, IIO, IIO Ring Port (IRP), PCU, PMON registers are accessed through x86 RD/WRMSR instructions. See Table 1-9.
- IMC PMON registers are accessed through the MMIO address space. The M2M, Intel® UPI, and the M3 Intel® UPI PMON registers are accessed through the PCI device configuration space.

Irrespective of the address-space difference and with only minor exceptions, the bit-granular layout of the control registers to program event code, unit mask (umask), start or stop, and signal filtering via threshold or edge detect are the same.

1.7 Addressing Uncore PMON State

The following is a list of registers provided in Sapphire Rapids server uncore for performance monitoring.

1.7.1 Uncore Performance Monitoring State in the MSR Space

As mentioned previously, the PMON blocks in the uncore have some number of paired counter or control (typically 4) registers, a unit status and unit control register. Many units may offer extra PMON state such as event filters or fixed counters.

Find the CHA filter MSR for a single instance next.

Table 1-8. Global Performance Monitoring Registers (MSR)

MSR Addresses	Descriptions
0x200E	CHA Filter

There are a number of free-running counters in each IIO stack that collect counts for I/O bandwidth for each port. The MSR addresses used to access that state are detailed in the following tables.The addresses to the other free-running counters in other boxes are calculated using a simple stride to the total number of boxes available.

Table 1-9. Free-running IIO Bandwidth "In" Counters in MSR Space

	Port 7 BW In	Port 6 BW In	Port 5 BW In	Port 4 BW In	Port 3 BW In	Port 2 BW In	Port 1 BW In	Port 0 BW In
M2IOSF 0	0x3807	0x3806	0x3805	0x3804	0x3803	0x3802	0x3801	0x3800

Table 1-10. Free-running IIO Bandwidth "Out" Counters in MSR Space

	Port 7 BW Out	Port 6 BW In	Port 5 BW In	Port 4 BW In	Port 3 BW In	Port 2 BW In	Port 1 BW In	Port 0 BW In
M2IOSF 0	0x380F	0x380E	0x380D	0x380C	0x380B	0x380A	0x3809	0x3808

Note: See each unit's performance monitoring section for any related state not covered here.

1.8 Uncore Performance Monitoring State in the PCICFG Space

As of Sapphire Rapids, there are a couple free-running counters and a fixed counter in each Memory Controller (MC) to collect counts for read and write bandwidth.

Each such block will have a PCICFG B:D:F and a device ID. The registers are presented as offsets to the PMON block's base address.

Table 1-11. IMC Fixed Counters

DCLK Ctr	DCKL Ctr
0x22838	0x22854

Table 1-12. IMC Free Running Counters

DCLK	rpq_active_cycles	wpq_active_cycles
0x22B0	0x2318	0x2320

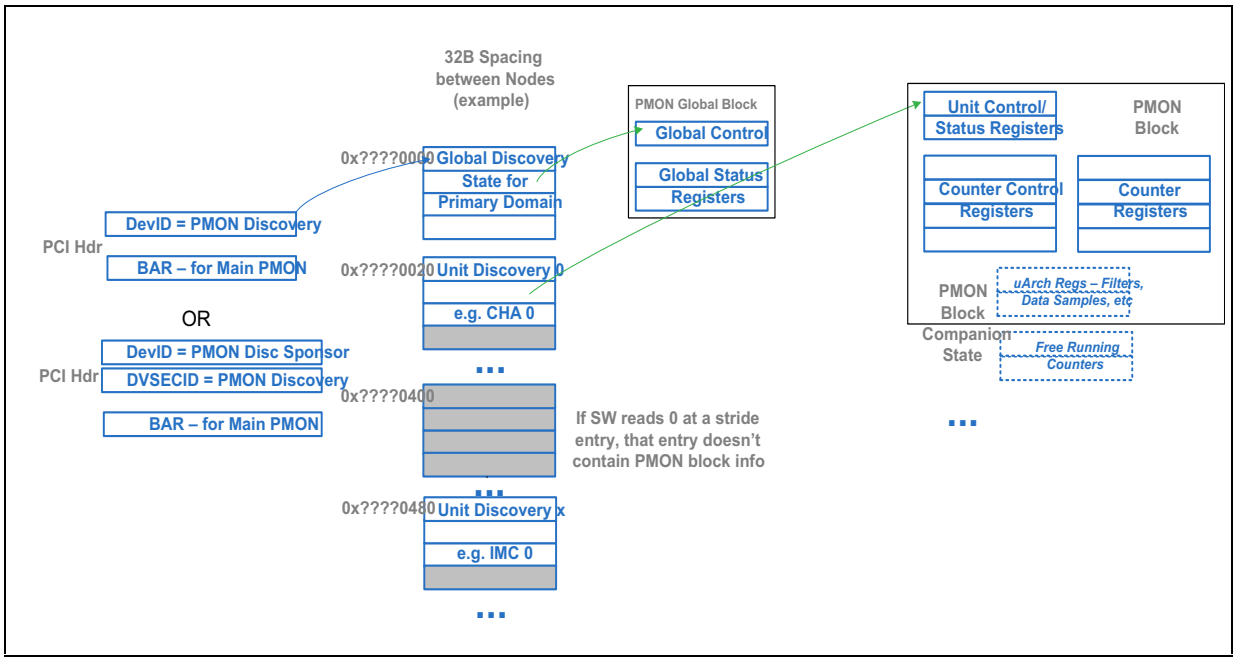
1.9 Introduction to Discovery - Self Describing HW

In Sapphire Rapids the self-describing HW starts by reading through an MMIO page worth of information, the SW can "discover" all the standard PMON registers in the global block followed by all the standard PMON registers in each of the units.

The non-standard PMON registers will not be included. For example, free running counters like the Mem/IIO BW counters, fixed counters like UCLK and Data Clock (DCLK) and extra filtering and matching registers such as the ones in the CHA and M2M. The SW tools that support these microarchitecture-specific extensions to the standard monitoring capabilities, will have to hardcode access as they had before.

The discovery roughly follows the basic uncore PerfMon* composition as illustrated in Figure 1-1.

Figure 1-11. Discovery - An Overview

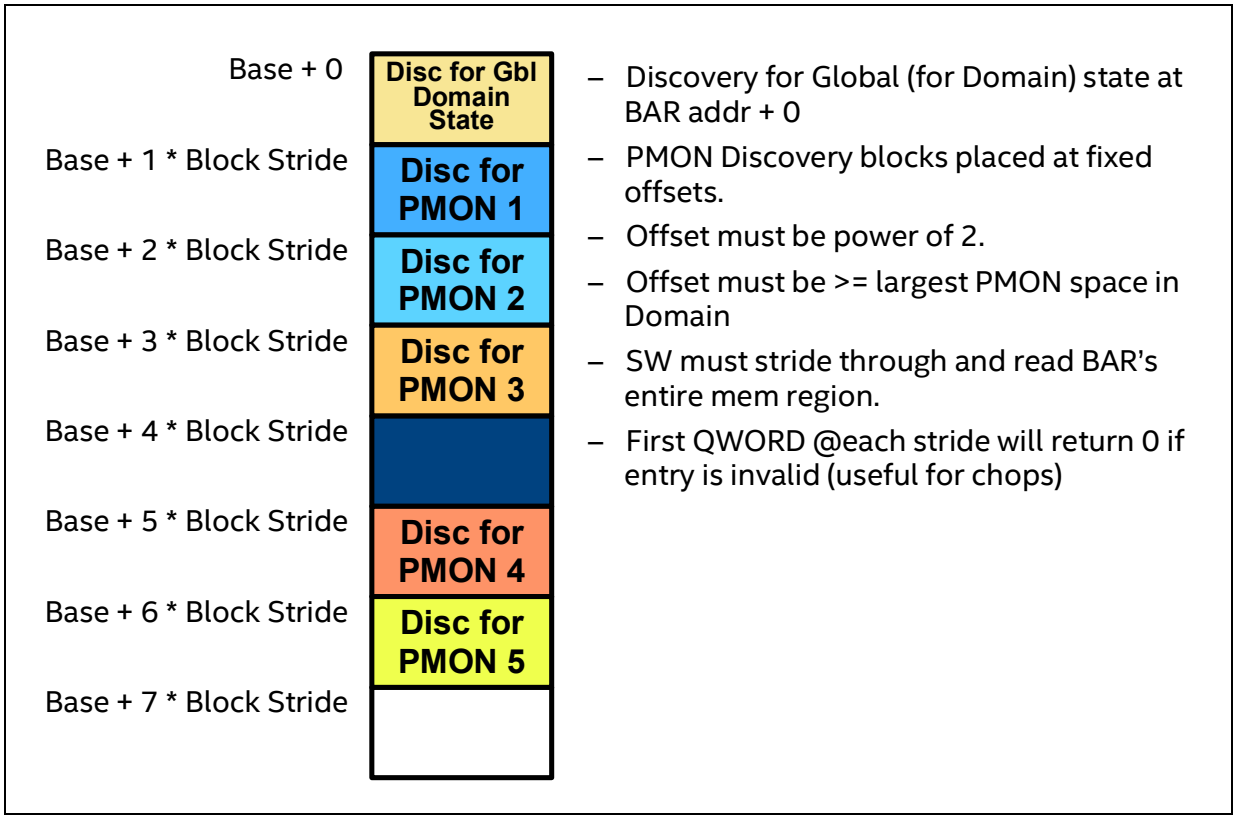


The first step is for the SW to find the device header sponsoring the MMIO page worth of the PMON discovery. To do this, the SW, while walking through the list of available PCI headers, hast to look for either the PCI header labeled "PMON discovery" or the Device Security Enhancements (DVSEC) substructure (for example, extended capability) labeled "PMON discovery". Once found, the SW can read the Base Address Register (BAR) and page size to determine the bounds of the discovery information.

Note: An example code is provided in Section 1.10.1.

The following diagram illustrates the basic structure of PMON discovery within the page. The SW has to first read the global discovery information from the offset 0x0, see Figure 1-13.

Figure 1-12. Discovery - Visual Guide for How the SW Strides the Page



In reading the global discovery, the SW can determine where the global control and status registers are, how large each block of the unit discovery information is (block stride), and the number of strides it will take to reach the end of the page (maximum blocks).

1.9.1 Global Discovery

The global entries in the PMON discovery page are to inform the SW:

- How to address the global control (global control address).
- How to address the other registers that form the global block.
- What address space these registers are accessed through.
- How to read through the rest of the discovery page to find all the unit discoveries.

Figure 1-13. Discovery - Global State

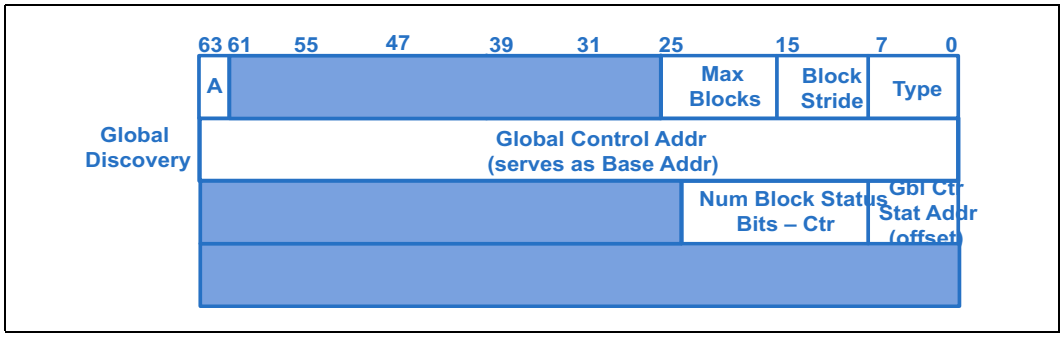


Table 1-13. Global Discovery– Field Definitions

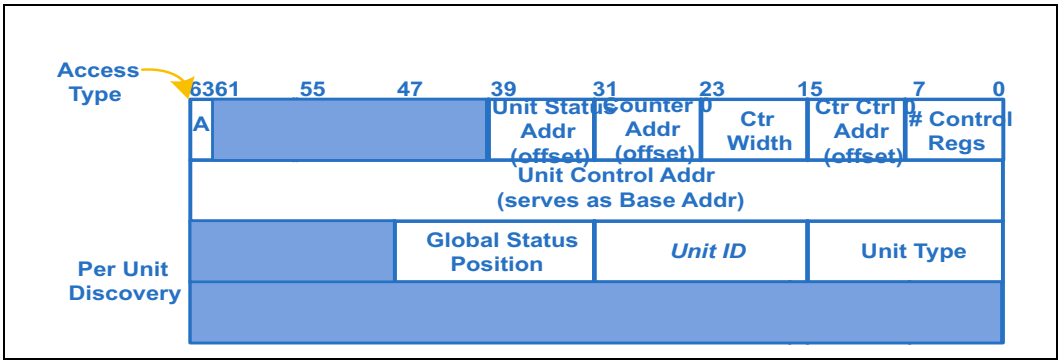
	Fields	Bits	Descriptions
Global node +0	Access type	63:61	The global state is accessed through: 00 - MSR space 01 - MMIO space 10 - PCICFG space
	rsv	**60:26**	**Reserved**
	Max blocks	25:16	The number of strides (0 for the global discovery node) the SW will need to make through the address space to ensure that all the unit discovery state from the domain has been found.
	Block stride	15:8	The length of each stride represents the amount of space reserved for each block of discovery. From the base address, the SW will need to stride through MaxBlocks-1 times from the base address to identify all discovery state.
	Type	7:0	Domain type
Global node +1	Global control address	63:0	Address to the global control register
Global node +2	**rsv**	**63:24**	**Reserved**
	Number block status address bits - counters	23:8	How many status bits are allocated to track overflows? For cases there are more than 64 status bits, the SW should divide this value by 64 to calculate the number of contiguously addressed counter status registers.
	Global Ctr status address (offset)	7:0	8b offset from global control address to first counter status register.

The SW can then stride the rest of the MMIO page to identify each PMON block. Any non-0 entry provides discovery information about a unit's PMON block.

```
For i = 0; i <= MaxBlocks - 1; i += BlockStride; {
        if page_of_discovery[i] != 0 process_unit_discovery()_
```

1.9.2 Unit Discovery

Figure 1-14. Discovery - Unit State



Each of the blocks of the unit discovery information tells the SW:

- The address space these registers are accessed through.
- How to address the global control (unit control address).
- Given the unit control's address, how to address the other standard registers in each PMON block - Includes counter control or counter pairs, the unit control, and the unit status registers.
- The "Unit ID" used to determine which of Sapphire Rapids event files is associated with this PMON block.

Table 1-14. Unit Discovery– Field Definitions (Sheet 1 of 2)

	Fields	Bits	Descriptions
Global node +0	Access type	63:61	The unit state is accessed through: 00 - MSR space 01 - MMIO space 10 - PCICFG space
	rsv	**60:40**	**Reserved**
	Unit status address (offset)	39:32	8b offset from the unit control address to the first unit status register.
	Counter 0 address (offset)	31:24	8b offset from the unit control address to the first counter register. Additional counters are contiguously spaced from first.
	Counter width	23:16	Number of bits in data register.
	Counter control 0 address (offset)	15:8	8b offset from unit control address to the first counter control register. Additional counter controls are contiguously spaced from first.
	Number control registers	7:0	Number of counter control registers paired with data registers in this unit.
Global node +1	Unit control address	63:0	Address to this unit control register.

Table 1-14. Unit Discovery– Field Definitions (Sheet 2 of 2)

	Fields	Bits	Descriptions
Global node +2	**rsv**	**63:48**	**Reserved**
	Global status position	47:32	16b field to tell the SW which bit in the global status belongs to the PMON block.
	Unit ID	31:16	Which number of this unit type? For cases where there are more than one instance of a particular unit, this identifies the specific PMON block for the unit type. For example, CHA #4 or IMC PMON block #2
	Unit type	15:0	What kind of unit is the PMON block associated with? Each unit of a unit type will offer the same event list. Each unit of a unit type will offer the same uarch specific PMON HW.

1.10 Guidance for the SW

1.10.1 Guidance on Finding PMON Discovery and Reading It

The following details the code to find the device sponsoring PMON discovery. It also shows how to address the MMIO page worth of discovery, traverse through it and find all the PMON registers.

```
/* Capability ID for discovery table device */
#define UNCORE_EXT_CAP_ID_DISCOVERY                    0x23
/* DVSEC offset */
#define UNCORE_DISCOVERY_DVSEC_OFFSET                  0x8
/* mask of DVSEC_ID */
#define UNCORE_DISCOVERY_DVSEC_ID_MASK                 0xffff
/* PMON discovery entry type ID */
#define UNCORE_DISCOVERY_DVSEC_ID_PMON                 0x1
/* mask of BIR */
#define UNCORE_DISCOVERY_DVSEC_BIR_MASK                0x7
/* discovery table size */
#define UNCORE_DISCOVERY_MAP_SIZE                      0x80000

struct uncore_global_discovery {
       union {
              u64    table1;
              struct {
                     u64    type : 8,
                            stride : 8,
                            max_units : 10,
                            __reserved_1 : 36,
                            access_type : 2;
              };
       };
       union {
```

```
                u64     table2;
                u64     global_ctl;
        };
        union {
                u64     table3;
                struct {
                u64     status_offset : 8,
                        num_status : 16,
                        __reserved_2 : 40;
                };
        };
};

struct uncore_unit_discovery {
        union {
                u64     table1;
                struct {
                u64     num_regs : 8,
                        ctl_offset : 8,
                        bit_width : 8,
                        ctr_offset : 8,
                        status_offset : 8,
                        __reserved_1 : 22,
                        access_type : 2;
                };
        };
        union {
                u64     table2;
                u64     box_ctl;
        };
        union {
                u64     table3;
                struct {
                        u64     box_type : 16,
                                box_id : 16,
                                __reserved_2 : 32;
                };
        };
};


/* Go through the entire PCI devices tree */
while ((dev = pci_get_device(PCI_VENDOR_ID_INTEL, PCI_ANY_ID, dev)) !=
NULL) {

        /* Walk the Extended Capability structures looking for a DVSEC
structure with unique capability ID 0x23 */
        while ((dvsec = pci_find_next_ext_capability(dev, dvsec,
UNCORE_EXT_CAP_ID_DISCOVERY))) {
```

```
                /* read the DVSEC_ID (15:0) */
                pci_read_config_dword(dev, dvsec +
UNCORE_DISCOVERY_DVSEC_OFFSET, &val);
                entry_id = val & UNCORE_DISCOVERY_DVSEC_ID_MASK;

                /* check if it is PMON discovery entry */
                if (entry_id == UNCORE_DISCOVERY_DVSEC_ID_PMON) {

                        /* read BIR value (2:0) */
                        pci_read_config_dword(dev, dvsec +
UNCORE_DISCOVERY_DVSEC_OFFSET + 4, &bir);
                        bir = bir & UNCORE_DISCOVERY_DVSEC_BIR_MASK;

                        /* calculate the BAR offset of global discovery table
*/
                        bar_offset = 0x10 + (bir * 4);

                        /* read the BAR address of global discovery table */
                        pci_read_config_dword(dev, bar_offset, &pci_dword);

                        /* Map whole discovery table */
                        addr = pci_dword & ~(PAGE_SIZE - 1);
                        io_addr = ioremap(addr, UNCORE_DISCOVERY_MAP_SIZE);

                        /* Read Global Discovery table */
                        memcpy_fromio(&global, io_addr, sizeof(struct
uncore_global_discovery));

                        /* Read Unit Discovery table one by one */
                        for (i = 0; i < global.max_units; i++) {
                                memcpy_fromio(&unit, io_addr + (i + 1) *
(global.stride * 8), sizeof(struct uncore_unit_discovery));

                                /* parse the unit discovery table  here*/
                        }
                }
        }
}
```

1.10.2 Guidance on Finding the Package's Bus Number for the Uncore PMON Registers in PCICFG Space

The PCI-based uncore units in Sapphire Rapids can be found using the bus, the device, and the functions numbers. However, the "bus no" has to be found dynamically in each package. The code is embedded next.

First, for each package, it is necessary to read the node ID offset in the Ubox. That read needs to match the Group Identifier (GID) offset of the Ubox in a specific pattern to get the "bus no" for the package. This "bus no" can then be used with the given the Device:Function (D:F) listed with each box's counters that are accessed through the PCICFG space.

ED: The one undefined piece in the following code is PCI_Read_Ulong, a function that simply reads the value from the PCI address. This function, or the ones similar to it, can be found in a more general PCI library (the composition of which is OS dependent).

Unfortunately, a link to a suitable version of the library was not readily available. The following are reference links to a comparable open source version of the library.

https://github.com/opcm/pcm/blob/master/ pci.h and pci.cpp

```
#define DRV_IS_PCI_VENDOR_ID_INTEL            0x8086
#define VENDOR_ID_MASK                        0x0000FFFF
#define DEVICE_ID_MASK                        0xFFFF00000
#define DEVICE_ID_BITSHIFT                    16

#define PCI_ENABLE                            0x80000000
#define FORM_PCI_ADDR(bus,dev,fun,off)        (((PCI_ENABLE))      | \
                                               ((bus & 0xFF) << 16)| \
                                               ((dev & 0x1F) << 11)| \
                                               ((fun & 0x07) << 8) | \
                                               ((off & 0xFF) << 0))


#define SPR_SERVER_SOCKETID_UBOX_DID   0x3250

//the below LNID and GID applies to Sapphire Rapids Server
#define UNC_SOCKETID_UBOX_LNID_OFFSET   0xC0
#define UNC_SOCKETID_UBOX_GID_OFFSET    0xD4

for (bus_no = 0; bus_no < 256; bus_no++) {
  for (device_no = 0; device_no < 32; device_no++) {
   for (function_no = 0; function_no < 8; function_no++) {

     // find bus, device, and function number for socket ID UBOX device
     pci_address = FORM_PCI_ADDR(bus_no, device_no, function_no, 0);
     value = PCI_Read_Ulong(pci_address);

     vendor_id = value & VENDOR_ID_MASK;
     device_id = (value & DEVICE_ID_MASK) >> DEVICE_ID_BITSHIFT;

     if (vendor_id != DRV_IS_PCI_VENDOR_ID_INTEL) {
       continue;
        }
     if (device_id == SPR_SERVER_SOCKETID_UBOX_DID) {
      // first get node id for the local socket
```

```
        pci_address = FORM_PCI_ADDR(bus_no, device_no, function_no,
                      UNC_SOCKETID_UBOX_LNID_OFFSET);
        gid = PCI_Read_Ulong(pci_address) & 0x00000007;

        // Get the node id mapping register:
        // Basic idea is to read the Node ID Mapping Register (below)
        // and match one of the nodes with gid that we read above
        // from the Node ID configuration register (above).
        // Every three bits in the Node ID Mapping Register maps to a
        // particular node (or package). Bits 2:0 maps to package 0,
        // bits 5:3 maps to package 1, and so on. Thus, we have to
        // parse every triplet of bits to find the match.

        pci_address = FORM_PCI_ADDR(bus_no, device_no, function_no,
                      UNC_SOCKETID_UBOX_GID_OFFSET);
        mapping = PCI_Read_Ulong(pci_address);


         for (i = 0; i < 8; i++){
          if (nodeid == ((mapping >> (3 * i)) & 0x7)) {
                gid = i;
                break;
          }
         }

         UNC_UBOX_package_to_bus_map[gid] = bus_no;
         }
    }
   }
}
```

1.10.3 Guidance on Resolving Addresses for Uncore PMON Registers in MMIO Space

The MMIO-based uncore units in Sapphire Rapids can be found by taking the Device ID and looking up the BAR (base address offset) that governs that unit's registers. For Sapphire Rapids, the BAR lookup is a two-step process as outlined next.

Once the base address has been resolved, simply add the published offsets to reference the PMON registers.

```
/* MMIO_BASE found at Bus U0, Device 0, Function 1, offset D0h. */
#define SPR_X_IMC_MMIO_BASE_OFFSET      0xd0
#define SPR_X_IMC_MMIO_BASE_MASK        0x1FFFFFFF
/* MEM0_BAR found at Bus U0, Device 0, Function 1, offset D8h. */
#define SPR_X_IMC_MMIO_MEM0_OFFSET      0xd8
#define SPR_X_IMC_MMIO_MEM_STRIDE       0xd04
#define SPR_X_IMC_MMIO_MEM_MASK        0x7FF
/*
```

```
* Each IMC has two channels.
* The offset starts from 0x22800 with stride 0x8000
*/
#define SPR_IMC_MMIO_CHN_OFFSET       0x22800
#define SPR_IMC_MMIO_CHN_STRIDE       0x8000
/* IMC MMIO size*/
#define SPR_X_IMC_MMIO_SIZE             0x4000


/*
* pkg_id: Socket id
* imc_idx: The IMC index
* channel_idx: The channel index
*/
Void *map_imc_pmon(int pkg_id, int imc_idx, int channel_idx)
{
       struct pci_dev *pdev = NULL;
       resource_size_t addr;
       u32 pci_dword;
       void *io_addr;
       int mem_offset;

/*
       * Device ID of Bus U0, Device 0, Function 1 is 0x3251 */
       * Get its pdev on the specific socket.
*/
       while(1){
              pdev = pci_get_device(PCI_VENDOR_ID_INTEL, 0x3251, pdev);
                            if ((!pdev) || (pdev->bus ==
UNC_UBOX_package_to_bus_map[pkg_id]))
                                          break;
               }
       if (!pdev)
              return NULL;

       /* read MEMn addr (51:23) from MMIO_BASE register */
       pci_read_config_dword(pdev, SPR_IMC_MMIO_BASE_OFFSET, &pci_dword);
       addr = (pci_dword & SPR_IMC_MMIO_BASE_MASK) << 23;

       /* read MEMn addr (22:12) from MEMn_BAR register */
        mem_offset = SPR_IMC_MMIO_MEM0_OFFSET + mem_idx *
SPR_IMC_MMIO_MEM_STRIDE;
       pci_read_config_dword(pdev, mem_offset, &pci_dword);
       addr |= (pci_dword & SPR_IMC_MMIO_MEM_MASK) << 12;

       /* IMC PMON registers start from PMONUNITCTRL */
       addr += SPR_IMC_MMIO_CHN_OFFSET + channel_idx *
SPR_IMC_MMIO_CHN_STRIDE;
```

```
        /* map the IMC PMON registers */
        io_addr = ioremap(addr, SPR_IMC_MMIO_SIZE);

        return io_addr;
}
```

1.10.4 Setting up a Monitoring Session

On the HW reset, all the counters are disabled. The enabling is hierarchical, so the following steps (which include programming the event control registers and enabling the counters to begin collecting events) must be taken to set up a monitoring session. Section 1.10.5 covers the steps to stop or re-start counter registers during a monitoring session.

A bug with the .rst logic. was pointed out due to clock gating, some of the boxes require that the counter level enable bits must be set before they can be reset. This means all these steps have to change and there is almost no possibility of two software entities using their own sampling intervals.

Global Settings in the UBox: (These are necessary for U-Box monitoring).

1. Freeze all the uncore counters by setting U_MSR_PMON_GLOBAL_CTL.frz_all to 1 **OR** if the box level freezes the control preferred:
2. Freeze the box counters while setting up the monitoring session. For example, set Cn_MSR_PMON_BOX_CTL.frz to 1.
3. Select the event to monitor if the event control register has not been programmed:
    a. Program the .ev_sel and .umask bits in the control register with the encoding necessary to capture the requested event along with any signal conditioning bits (.thresh/.edge_det/.invert) used to qualify the event.

Back to the box level:

4. Reset the counters in each box to ensure no stale values have been acquired from previous sessions. Resetting the control registers, particularly those that will not be used, is also recommended if for no other reason than to prevent errant overflows.

    To reset both the counters and control registers write the following registers:

Note: This is based on the idea that .unfrz_all is set to 1 during chip the bring up such that the UBox global control will no longer need to be touched and can be hidden from customers.

    a. For each CHAx, set Cn_MSR_PMON_UNIT_CTL[9:8] to 0x300
    b. For each Modular Die Fabric (MDF)x, set Cn_MSR_PMON_UNIT_CTL[9:8] to 0x300
    c. For each DRAM Channel, set MCn_CHy_PCI_PMON_UNIT_CTL[9:8] to 0x300
    d. Set PCU_MSR_PMON_UNIT_CTL[9:8] to 0x300
    e. For each Intel® UPI Link, set M3_Ly_PCI_PMON_UNIT_CTL[9:8] to 0x300
    f. For each Intel® UPI Link, set UPI_Ly_PCI_PMON_UNIT_CTL[9:8] to 0x300
    g. For each IIO stack, set M2n_PCI_PMON_UNIT_CTL[9:8] to 0x300
    h. For each IIO stack, set IIOn_MSR_PMON_UNIT_CTL[9:8] to 0x300
    i. For each IIO stack, set IRPn_MSR_PMON_UNIT_CTL[9:8] to 0x300

5. Select how to gather data.
    a. If polling: Skip to f.
    b. If sampling: To set up a sample interval, the software can preprogram the data register with a value of (2^[register bit width - up to 48] - sample interval length). Doing so allows software, through use of the PMI mechanism, to be notified when the number of events in the sample have been captured. Capturing a performance monitoring sample every "X cycles" (the fixed counter in the UBox counts uncore clock cycles) is a common use of this mechanism. For example, to stop counting and receive notification when the 1,000,000th data flit is transmitted from Intel® UPI on link 0.
        — Set UPI_L0_PCI_PMON_CTR1 to (2^48- 1000)
        — Set UPI_L0_PCI_PMON_CTL1.ev_sel to 0x2
        — Set UPI_L0_PCI_PMON_CTL1.umask to 0xF
        — Set U_MSR_PMON_GLOBAL_CTL.pmi_core_sel to which core the monitoring thread is executing on
    c. Enable counting at the global level by setting the U_MSR_PMON_GLOBAL_CTL.frz bit to 0. **OR**
6. Enable counting at the box level by unfreezing the counters in each box. For example, set Cn_MSR_PMON_BOX_CTL.frz to 0

And with that, the counting will begin.

1.10.5 Reading the Sample Interval

The software can poll the counters whenever it chooses, or wait to be notified that a counter has overflowed (by receiving a PMI).

- **Polling**: Before reading, it is recommended that the software freezes the counters at either the global level (U_MSR_PMON_GLOBAL_CTL.frz_all) or in each box with active counters (by setting *_PMON_UNIT_CTL.frz to 1). After reading the event counts from the counter registers, the monitoring agent can choose to reset the event counts to avoid an event-count wrap-around; or resume the counter register without resetting their values. The latter choice will require the monitoring agent to check and adjust for potential wrap-around situations.
- **Frozen counters**: If the software sets the counters to freeze on overflow and sends notification when it happens, the next question is: Who caused the freeze?

The overflow bits are stored hierarchically within the Sapphire Rapids uncore. First, the software hast to read the U_MSR_PMON_GLOBAL_STATUS.ov_* bits to determine which boxes sent an overflow. Then read that box *_PMON_GLOBAL_STATUS.ov field to find the overflowing counter.

Note: More than one counter may overflow at any given time. Certain boxes may have more than one PMON block (For example, the IMC has a PMON block in each channel). It may be necessary to read all STATUS registers in the box to determine which counter overflowed.

1.10.6 Enabling a New Sample Interval from Frozen Counters

- **Clear all uncore counters**: For each box in which counting occurred, set *_PMON_BOX_CTL.rst_ctrs to 1. The global reset signal is broken and not to be fixed.
- **Clear all overflow bits**: This includes clearing U_MSR_PMON_GLOBAL_STATUS.ov_* as well as any *_BOX_STATUS registers that have their overflow bits set.
  - For example, if the counter 3 in Intel® UPI Link 1 overflowed, the software has to set UPI_L1_PCI_PMON_BOX_STATUS.ov[3] to 1 to clear the overflow.
- **Create the next sample**: Reinitialize the sample by setting the monitoring data register to (2^48 - sample_interval). Or set up a new sample interval as outlined in Section 1.10.4.
- **Re-enable counting**: Set U_MSR_PMON_GLOBAL_CTL.frz_all to 0.

2 Sapphire Rapids Uncore Performance Monitoring

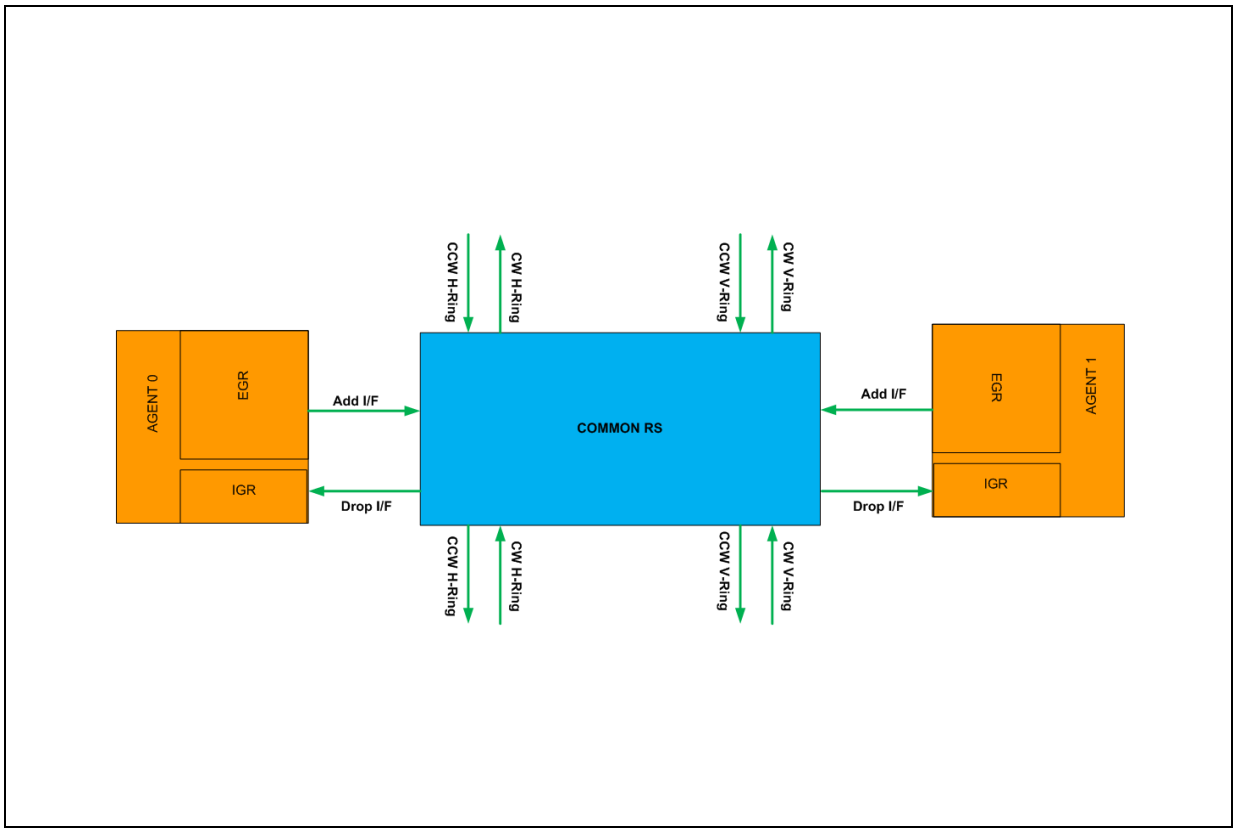
2.1 Mesh Performance Monitoring

For all the boxes that must communicate with the mesh, there are a common set of events to capture various kinds of information about traffic flowing through their connection to the mesh. The same encodings are used to request the mesh events in each box.

This common mesh stop event list is available in the CHA, M2PCIe* and M3 Intel® UPI.

Note: The common mesh stop events for the M2M would have an additional bit enabled for its programming. The event list is available in the M2M section.

Figure 2-1. Uncore PMON Components and Hierarchy



2.1.1 Mesh Performance Monitoring Events

There are events to track information related to all traffic passing through each box connection to the mesh.

- Credit tracking and stalls due to lack of credits
- Credits are required for each row (for example, transgress) destination through either side of the mesh stop for each path (Address [AD] and Block [BL])
- Mesh stop events
- To track the ingress or egress traffic, mesh utilization (broken down by direction and ring type), bypass statistics and more
- Bounce and starvation events
- Events to help recognize when the mesh is becoming or is saturated.

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.2 CHA Performance Monitoring

The Last Level Cache (LLC) coherence engine and CHA merges the caching agent and Home Agent (HA) responsibilities of the chip into a single block. In its capacity as a caching agent, the CHA manages the interface between the core, the IIO devices, and the LLC. In its capacity, as a HA, the CHA manages the interface between the LLC and the rest of the Intel® UPI coherent fabric as well as the on die memory controller.

All core and IIO DMA transactions that access the LLC are directed from their source to a CHA via the mesh interconnect. The CHA is responsible for managing data delivery from the LLC to the requester and maintaining coherence between the all the cores and IIO devices within the socket that share the LLC. It is also responsible for generating snoops and collecting snoop responses from the local cores when the MESIF protocol requires it.

Similarly, all incoming traffic from remote sockets that maps to the socket's local memory are directed from the Intel® UPI links to a CHA via the mesh interconnect. The CHA is responsible for managing the coherence across all sockets in the system for the socket's memory following the protocols defined in the Intel® UPI Specification. It manages directory state for the local memory, conflicts, and memory ordering rules for such requests.

In the process of maintaining the cache coherency within the socket, and across the system in a multi-socket system, the CHA is the gate keeper for all Intel® UPI messages that have addresses mapping to the socket's memory, as well as the originator of all Intel® UPI messages that originate from the cores within its socket when attempts are made to access memory in another socket. It is responsible for ensuring that all Intel® UPI messages that pass through the socket remain coherent.

The CHA can manage a large number of simultaneous requests in parallel, but in order to maintain proper memory ordering it does ensure that whenever multiple incoming requests to the same address are pending (whether they originated from a core or IIO device within the socket or came in from another socket through one of the Intel® UPI links) only one of those requests is being processed at a time.

Since the LLC cache is not inclusive of the Intel® architecture cores internal caches, the total cache capacity of the socket is much larger than the LLC capacity and each CHA is responsible for monitoring a portion of that available Intel® architecture core cache capacity for the purpose of maintaining coherence between the Intel® architecture core caches and the rest of the Intel® UPI coherent fabric.

Every physical memory address in the system is uniquely associated with a single CHA instance, via a proprietary hashing algorithm, that is designed to keep the distribution of traffic across the CHA instances relatively uniform for a wide range of possible address patterns. This enables the individual CHA instances to operate independently, each managing its slice of the physical address space without any CHA in a given socket ever needing to communicate with the other CHAs in that same socket.

2.2.1 CHA Performance Monitoring Overview

Each of the CHAs in the Sapphire Rapids uncore supports event monitoring through four 48-bit wide counters (Cn_MSR_PMON_CTR{3:0}). With but a small number of exceptions, each of these counters can be programmed (Cn_MSR_PMON_CTL{3:0}) for any available event.

Note: Occupancy events can only be measured in the counter 0.

2.2.1.1 Special Note on CHA Occupancy Events

Although only counter 0 supports occupancy events, it is possible to program counters 1 to 3 to monitor the same occupancy event by selecting the "OCCUPANCY_COUNTER0" event code on counters 1 to 3.

This allows:

- **The thresholding on all four counters:** While no more than one queue can be monitored at a time, it is possible to setup different queue occupancy thresholds on each of the four counters. For example, if one wanted to monitor the Interrupt Request (IRQ), one could setup thresholds of 1, 7, 14, and 18 to get a picture of the time spent at different occupancies in the IRQ.
- **Average latency and average occupancy:** It can be useful to monitor the average occupancy in a queue as well as the average number of items in the queue. An option is to program the counter 0 to accumulate the occupancy, counter 1 with the queue's allocations event, and counter 2 with the OCCUPANCY_COUNTER0 event, and a threshold of 1. The latency can then be calculated by counter 0, counter 1, and occupancy by counter 0, counter 2.

2.2.2 Additional CHA Performance Monitoring

2.2.2.1 CHA PMON Counter Control - Difference from the Baseline

The CHA performance monitoring control registers provide a small amount of additional functionality. The following table defines those cases.

Figure 2-2. CHA Counter Control Register for Sapphire Rapids Server

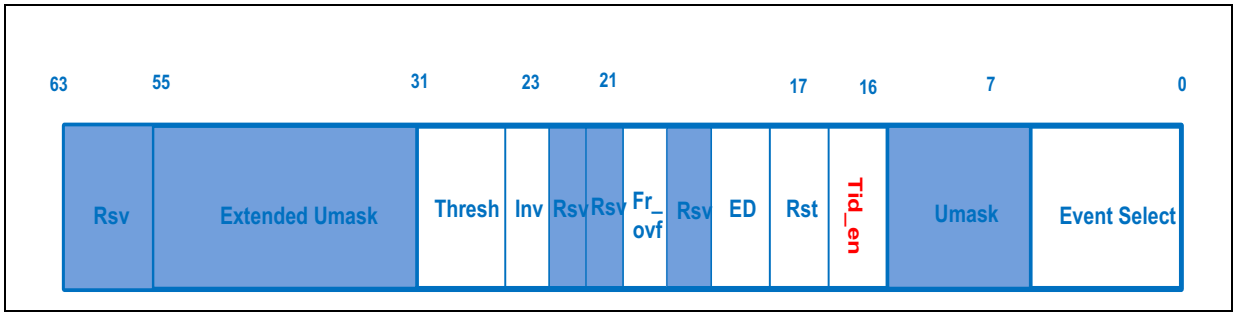


Table 2-1. Cn_MSR_PMON_CTL{3-0} Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Description
tid_en	16	RW-V	0	TID filter enable

2.2.2.2 CHA Filter Registers (Cn_MSR_PMON_BOX_FILTER0)

Any of the CHA events may be filtered by thread and core-ID. To do so, the control register *.tid_en* bit must be set to 1 and the .tid field in the FILTER register filled out. UPI_CREDITS may be filtered by link.

Figure 2-3. CHA PMON Filter Register

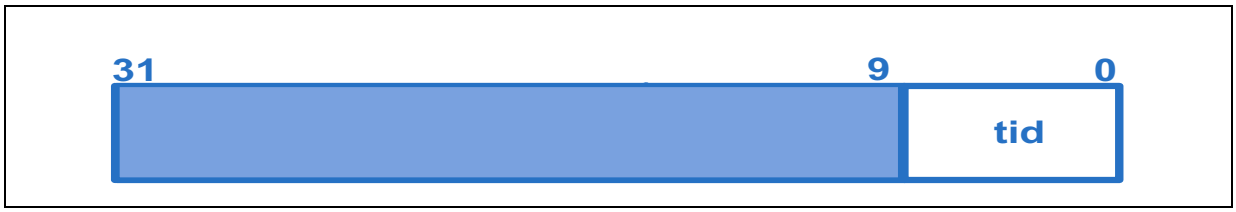


Table 2-2. Cn_MSR_PMON_BOX_FILTER Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Description
rsv	**31:10**	**RV**	**0**	**Reserved** **The SW must set to 0, or else behavior is undefined**
tid	9:0	0	0	[9:3] Core-ID [2:0] Thread 3-0 When the .tid_en is 0; the specified counter will count all events. To filter on a specific logical core, set the Core-ID to the desired core number and set the TID field to the desired thread. To filter on a source or destination other than an Intel® architecture core, set core-ID to one of the following and set TID to 0

2.2.3 CHA Performance Monitoring Events

The performance monitoring events within the CHA include all events internal to the LLC and HA as well as the events which track mesh related activity at the CHA and the core mesh stops, see Section 2.1.1 for the available mesh stop events.

The CHA performance monitoring events can be used to track the LLC access rates, the LLC hit or miss rates, the LLC eviction and fill rates, the HA access rates, the HA conflicts, and to detect evidence of back pressure on the internal CHA pipelines. In addition, the CHA has performance monitoring events for tracking the MESIF state transitions that occur as a result of data sharing across sockets in a multi-socket system.

Every event in the CHA is from the point of view of the CHA and it is not associated with any specific core since all cores in the socket send their LLC transactions to all CHAs in the socket. However, the Sapphire Rapids CHA provides a thread identification field in the Cn_MSR_PMON_BOX_FILTER register, which can be applied to the CHA events, to obtain the interactions between specific cores and threads.

There are separate sets of counters for each CHA instance. For any event, to get an aggregate count of that event for the entire LLC, the counts across the CHA instances must be added together. The counts can be averaged across the CHA instances to get a view of the typical count of an event from the perspective of the individual CHAs. Individual per-CHA deviations from the average can be used to identify hot-spotting across the CHAs or other evidences of non-uniformity in LLC behavior across the CHAs. Such hot-spotting is rare, though a repetitive polling on a fixed physical address is one obvious example of a case where an analysis of the deviations across the CHAs would indicate hot-spotting.

2.2.3.1 Acronyms Frequently Used in CHA Events

The Rings:

- **AD ring:** The core R/W requests and Intel® UPI snoops. The AD ring carries Intel® UPI requests and snoop responses from the C to Intel® UPI.
- **Block or Data (BL) ring:** The data is equal to 2 transfers for one cache line.
- **Acknowledge (AK) ring:** It acknowledges Intel® UPI to CHA and CHA to core. It carries snoop responses from the core to the CHA.
- **Invalidate (IV) ring:** The CHA snoop requests of core caches.

2.2.3.2 Key Queues

- **IRQ:** Requests from Intel® architecture cores
- **IPQ:** Ingress Probe Queue on AD Ring. They are remote socket snoops sent from Intel® UPI LL.
- **ISMQ:** Ingress Subsequent Messages (response queue). They are associated with message responses to ingress requests (For example, data responses, Intel® UPI completion messages, core snoop response messages and the “GO” reset queue).
- **PRQ:** Requests from the IIO.
- **RRQ:** Remote Request Queue. They are remote socket read requests, from Intel® UPI to the local HA.
- **WBQ:** Writeback Queue. They are remote socket write requests, from UP to the local HA.

- **TOR:** Table Of Requests. They are the tracks pending CHA transactions.
- **RxC (also known as Ingress [IGR]) /TxC (also known as Egress [EGR]):** They are the ingress requests from the cores (via Common Mesh Stop [CMS]), and the egress requests headed for the mesh (via CMS) queues.

2.2.4 CHA Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the CHA box.

CLOCKTICKS

- **Title:**
- **Category:** Clocktick events
- **Event Code:** 0x1
- **Maximum Increments per Cycle (Inc/Cyc):** 1
- **Register Restrictions:** 0-3
- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.3 IMC Performance Monitoring

The Sapphire Rapids IMC provides the interface to the DRAM and communicates to the rest of the uncore through the M2M block.

The memory controller also provides a variety of RAS features such as ECC, memory access retry, memory scrubbing, thermal throttling, mirroring, and rank sparing.

2.3.1 Functional Overview

The memory controller communicates to the DRAM, translating R/W commands into specific memory commands and schedules them with respect to memory timing. The other main function of the memory controller is advanced ECC support.

2.3.2 IMC Performance Monitoring Overview

The IMC supports event monitoring through four 48-bit wide counters (MC_CHy_PCI_PMON_CTR{3:0}) and one fixed counter (MC_CHy_PCI_PMON_FIXED_CTR) for each DRAM channel (of which there are 2 in Sapphire Rapids) the MC is attached to. Each of these counters can be programmed (MC_CHy_PCI_PMON_CTL{3:0}) to capture any MC event.

2.3.3 Additional IMC Performance Monitoring

Following is a counter that always tracks the number of DRAM clocks in the IMC. The DCKL never changes frequency (on a given system), and therefore is a good measure of wall clock (unlike the uncore clock, which can change frequency based on system load).

.

Figure 2-4. PMON Control Register for DCLK

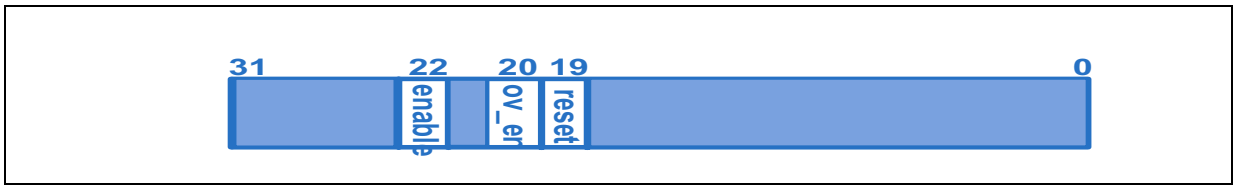


Table 2-3. MC_CHy_PCI_PMON_FIXED_CTL Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**31:23**	**RV**	**0**	**Ignored**
en	22	RW-V	0	Local counter enable
ig	**21**	**RV**	**0**	**Ignored**
ov_en	20	RW-V	0	When this bit is asserted and the corresponding counter overflows, a PMI exception is sent to the UBox.
rst	19	WO	0	When set to 1, the corresponding counter will be cleared to 0.
ig	**18:0**	**RV**	**0**	**Ignored**

Table 2-4. MC_CHy_PCI_PMON_CTR{FIXED,3-0} Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**63:48**	**RV**	**0**	**Ignored**
event_count	47:0	RW-V	0	48-bit performance event counter

There are a few free-running counters, providing information highly valuable to a wide array of customers, in each IMC that collect counts for cumulative R/W bandwidth across all channels.

"Free Running" counters cannot be changed by the SW operating in a normal environment. The SW cannot write them, stop them, nor can it reset the values.

Note: The counting will be suspended when the MC is powered down.

DDR CYCLES: There is one register per stack to track the number of DDR cycles as measured by the MC.

Table 2-5. MC_MMIO_PMON_FRCTR_DCLK Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**63:48**	**RV**	**0**	**Ignored**
event_count	47:0	RO-V	0	48-bit running count of DDR clocks captured in MC

WPQ ACTIVE CYCLES: They count the number of cycles the WPQ was used over the total number of DDR cycles.

Table 2-6. MC_MMIO_PMON_FRCTR_WPQ_ACTIVE Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**63:48**	**RV**	**0**	**Ignored**
event_count	47:0	RO-V	0	48-bit running count of data bytes read from the attached DIMM

RPQ ACTIVE CYCLES: They count the number of cycles the RPQ was utilized over the total number of DDR cycles.

Table 2-7. MC_MMIO_PMON_FRCTR_RPQ_ACTIVE Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Description
ig	**63:48**	**RV**	**0**	**Ignored**
event_count	47:0	RO-V	0	48-bit running count of data bytes read from the attached DIMM

2.3.4 IMC Performance Monitoring Events

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the IMC Box.

CLOCKTICKS

- **Title:**
- **Category:** DCLK events
- **Event Code:** 0x1
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:** 0-3
- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.4 IIO Performance Monitoring

The IIO stacks are responsible for managing the traffic between the PCI Express* (PCIe*) domain and the mesh domain. The IIO PMON block is situated near the IIO stacks traffic controller capturing the traffic controller as well as the PCIe* root port information. The traffic controller is responsible for translating the traffic coming in from the mesh (through M2IAL) and processed by the IRP into the PCIe* domain to the I/O agents, such as CBDMA, DMA and PCIe*.

2.4.1 IIO Performance Monitoring Overview

Each IIO Box, which sits near the IIO stack's traffic controller, supports event monitoring through four 48b-wide counters (IIO{5-0}_MSR_PMON_CTR/CTL{3:0}). Each of these counters can be programmed to count any IIO event.

2.4.2 Additional IIO Performance Monitoring

2.4.2.1 IIO PMON Counter Control - Difference from Baseline

IIO performance monitoring control registers provide a small amount of additional functionality. The following table defines those cases.

Figure 2-5. IIO Counter Control Register for Sapphire Rapids Server

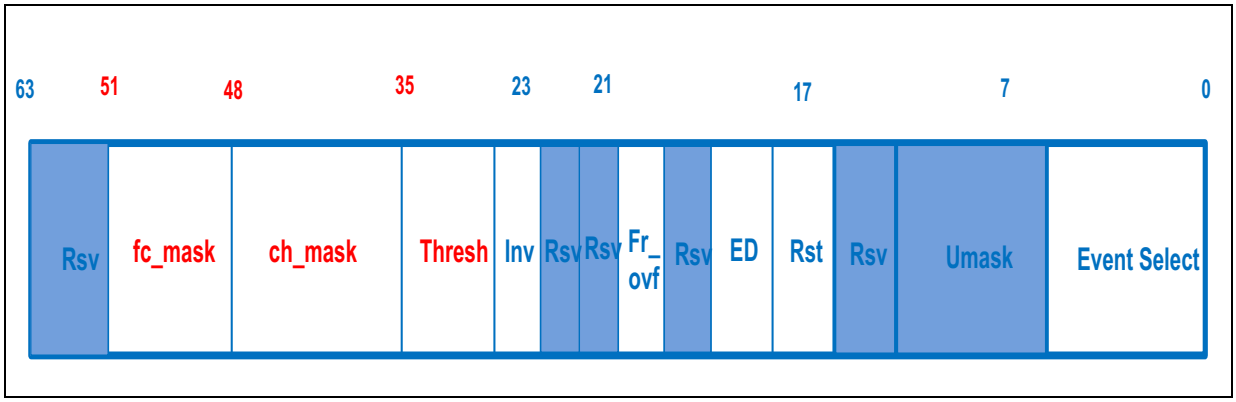


Table 2-8. IIOn_MSR_PMON_CTL{3-0} Register – Field Definitions (Sheet 1 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	63:51	RV	0	Reserved The SW must write to 0 else behavior is undefined.

Table 2-8. IIOn_MSR_PMON_CTL{3-0} Register – Field Definitions (Sheet 2 of 2)

Fields	Bits	Attributes	HW Reset Values	Descriptions
fc_mask	50:48	RW-V	0	FC Mask - applicable to certain events (Filter - fc) 0 - Posted requests 1 - Non-posted requests 2 - Completions
ch_mask	47:36	RW-V	0	Channel mask filter - applicable to certain events (Filter - channel)
thresh	35:24	RW-V	0	Threshold used in counter comparison

There are a number of free-running counters, providing information highly valuable to a wide array of customers, in each IIO Stack that collect counts for I/O bandwidth for each port.

"Free Running" counters cannot be changed by SW operating in a normal environment. The SW cannot write them, cannot stop them and cannot reset the values.

Note: Counting will be suspended when the IIO stack is powered down. There is one register per stack to track the number of IIO cycles.

Table 2-9. IIO_MSR_PMON_FRCTR_IOCLK Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**63:48**	**RV**	**0**	**Ignored**
event_count	47:0	RO-V	0	48-bit running count of I/O clocks

Inbound (PCIe* -> CPU) bandwidth: Counts DWs (4 bytes) of data, associated with writes and completions, transmitted from the I/O stack to the traffic controller.

Table 2-10. IIO_MSR_PMON_FRCTR_BW_IN_P{0-7} Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
ig	**63:36**	**RV**	**0**	**Ignored**
event_count	47:0	RO-V	0	48-bit running count of data bytes transmitted from link for this port.

2.4.3 IIO Performance Monitoring Events

The I/O provides events to track information related to all the traffic passing through its boundaries.

- Bandwidth consumed and transactions processed broken down by the transaction type
- Per-port utilization
- Link power states
- Completion buffer tracking

2.4.4 IIO Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the IIO Box.

CLOCKTICKS

- **Title:**
- **Category:** CLOCK events
- **Event Code:** 0x1
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:** 0-3
- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.5 IRP Performance Monitoring

The IRP is responsible for maintaining coherency for the IIO traffic targeting coherent memory.

2.5.1 IRP Performance Monitoring Overview

Each IRP box supports event monitoring through two 48b-wide counters (IRP{5-0}_MSR_PMON_CTR/CTL{1:0}). Each of these counters can be programmed to count any IRP event.

2.5.2 IRP Performance Monitoring Events

The IRP provides events to track information related to all the traffic passing through its boundaries.

- Write cache occupancy
- Ingress or egress traffic - by ring type
- Stalls awaiting credit
- Fire and Forget (FAF) queue

2.5.3 IRP Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the IRP Box.

CLOCKTICKS

- **Title:**
- **Category:** CLOCK events
- **Event Code:** 0x1
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:** 0-1

- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.6 Intel® UPI Link Layer Performance Monitoring

Sapphire Rapids uses a coherent interconnect for scaling to multiple sockets known as Intel® UPI. Intel® UPI technology provides a cache coherent socket to socket external communication interface between processors. The processor implements up to four Intel® UPI links depending on the specific product.

Intel® UPI is also used as a coherent communication interface between processors and the OEM Extensible Node Controllers* (XNC*).

There are up to three Intel® UPI agents, each with its own mesh stop. These links can be connected to a single destination (such as in a Downstream Port [DP]), or it can be connected to two separate destinations (4s Ring or DP) or multiple destinations. Therefore, it will be necessary to count Intel® UPI statistics for each agent separately.

The Intel® UPI module supports one Intel® UPI link (per mesh stop) and it is comprised of the following layers for each Intel® UPI link:

- **Physical Layer (PHY):** The Intel® UPI PHY is a hardware layer that lies between the PHY above it, and the physical wires that connect to other devices. The PHY is further sub-divided into the logical and electrical sub-blocks.
- **Link Layer:** The Intel® UPI link layer bi-directionally converts between protocol layer messages and the link layer flits, it passes them through shared buffers, and manages the flow control information per virtual channel. The link layer also detects errors and retransmits the packets on errors.
- **Routing Layer:** The routing layer is distributed among all agents that sends Intel® UPI messages on the mesh (Intel® UPI, CHA, PCIe*, IMC). The Intel® UPI module provides a routing function that determines the correct mesh stop from which to forward a given packet.
- **Protocol Layer:** The Intel® UPI module does not implement the protocol Layer. A protocol agent is a proxy for some entity which injects, generates, or services Intel® UPI transactions, such as memory requests, interrupts, and so on.
  - The protocol layer is implemented in the following modules:
    - CHA
    - PCIe*
    - CA (Ubox)
  - A CA in the CHA generates both requests and services snoops. A HA in the CHA services requests, generates snoops, and resolves conflicts. The CHA will sometimes behave as CA, sometimes as a HA, and sometimes it will behave as both at the same time. The PCIe* module handles most I/O proxy responsibilities.
  - The Ubox handles internal configuration space and some other interrupt and messaging flows. A HA acts as a proxy for the DRAM, while the PCIe* and Ubox handle all the non-DRAM (NCB and NCS) requests.

The Intel® UPI Link Layer is responsible for packetizing requests from the caching agent on the way out to the system interface. The Intel® UPI link layer processes information at a flit granularity.

A single Intel® UPI flit can pack up to three mesh packets in three slots. The Intel® UPI link layer has the ability to transmit up to three mesh packets per cycle in each direction. . It is not possible to monitor Rx and Tx flit information at the same time on the same counter.

Note: Flit slots are not symmetric in their ability to relay flit traffic. Any analysis of Intel® UPI BW must keep this in mind.

2.6.1 Intel® UPI Performance Monitoring Overview

Each Intel® UPI link supports event monitoring through four 48b-wide counters (U_Ly_PCI_PMON_CTR/CTL{3:0}). Each of these four counters can be programmed to count any Intel® UPI event.

2.6.2 Additional Intel® UPI Performance Monitoring

2.6.2.1 Intel® UPI Extra Registers - Companions to PMON HW

Intel® UPI box includes three registers that provide performance monitoring related information outside of the normal PMON infrastructure.

- A register that provides the current Intel® UPI transfer rate
- A 32b-free running counter that counts the number of cycles when the Rx side of the link is idle. It includes null cycles and cycles where the link is in L1 (for example powered down).
- A 32b-free running counter that counts the number of cycles where the Rx side of the link is in LLR.

Table 2-11. UPI_RATE_STATUS Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
rsv	**31:3**	**RV**	**0**	**Reserved** **The SW must write to 0, else behavior is undefined.**
UPI_rate	2:0	RO-V	11b	Intel® UPI rate This reflects the current Intel® UPI rate setting into the PLL.

Table 2-12. U_Ly_PCI_PMON_LINK_IDLE Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
event_count	31:0	RW-V	0	32-bit performance event counter

Table 2-13. U_Ly_PCI_PMON_LINK_LLR Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
event_count	31:0	RW-V	0	32-bit performance event counter

2.6.3 Intel® UPI LL Performance Monitoring Events

The Intel® UPI link layer provides events to gather information on topics such as:

- Tracking incoming (mesh bound) outgoing (system bound) transactions.

2.6.4 Intel® LL Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the IRP Box.

CLOCKTICKS

- **Title:**
- **Category:** CLOCK events
- **Event Code:** 0x1
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:** 0-3
- **Definition:** Counts the number of clocks in the Intel® UPI LL.

RxL_FLITS

- **Title:**
- **Category:** Flit events
- **Event Code:** 0x3
- **Maximum Inc/Cyc:** 3
- **Register Restrictions:** 0-3
- **Definition:** Shows legal flit time (hides impact of L0p and L0c).

Table 2-14. Unit Masks for RxL_FLITS

Extensions	umask [15:8]	Descriptions
SLOT0	bxxxxxxx1	Slot 0 Count Slot 0 - Other mask bits determine types of headers to count.
SLOT1	bxxxxxx1x	Slot 1 Count Slot 1 - Other mask bits determine types of headers to count.
SLOT2	bxxxxx1xx	Slot 2 Count Slot 2 - Other mask bits determine types of headers to count.

Table 2-14. Unit Masks for RxL_FLITS

Extensions	umask [15:8]	Descriptions
DATA	bxxxx1xxx	Data Count data flits (which consume all slots), but how much to count is based on slot 0-2 mask, so count can be 0 to 3, depending on which slots are enabled for counting.
ALL_DATA	b00001111	All data
LLCRD	bxxx1xxxx	LLCRD not empty Enables counting of LLCRD (with non-zero payload). This only applies to slot 2 since LLCRD is only allowed in slot 2.
NULL	bxx1xxxxx	Slot NULL or LLCRD empty LLCRD with all zeros is treated as NULL. Slot 1 is not treated as NULL if slot 0 is a dual slot. This can apply to slot 0, 1, or 2.
ALL_NULL	b00100111	Null flits received from any slot.
LLCTRL	bx1xxxxxx	LLCTRL Equivalent to an idle packet Enables counting of slot 0 LLCTRL messages.
IDLE	b01000111	Idle
PROTHDR	b1xxxxxxx	Protocol header Enables count of protocol headers in slot 0, 1, 2 (depending on slot umask bits)
NON_DATA	b10010111	All non data

TxL_FLITS

- **Title:**
- **Category:** Flit events
- **Event Code:** 0x2
- **Maximum Inc/Cyc:** 3
- **Register Restrictions:** 0-3
- **Definition:** Shows legal flit time (hides impact of L0p and L0c).

Table 2-15. Unit Masks for TxL_FLITS

Extensions	umasks [15:8]	Descriptions
SLOT0	bxxxxxxx1	Slot 0 Count Slot 0 - Other mask bits determine types of headers to count.
SLOT1	bxxxxxx1x	Slot 1 Count Slot 1 - Other mask bits determine types of headers to count.
SLOT2	bxxxxx1xx	Slot 2 Count Slot 2 - Other mask bits determine types of headers to count.
DATA	bxxxx1xxx	Data Count data flits (which consume all slots), but how much to count is based on slot 0-2 mask, so count can be 0-3 depending on which slots are enabled for counting.
ALL_DATA	b00001111	All data

Table 2-15. Unit Masks for TxL_FLITS

Extensions	umasks [15:8]	Descriptions
LLCRD	bxxx1xxxx	LLCRD Not empty Enables counting of LLCRD (with non-zero payload) This only applies to slot 2 since LLCRD is only allowed in slot 2
NULL	bxx1xxxxx	Slot NULL or LLCRD empty LLCRD with all zeros is treated as NULL. Slot 1 is not treated as NULL if slot 0 is a dual slot. This can apply to slot 0, 1, or 2.
ALL_NULL	b00100111	Idle
LLCTRL	bx1xxxxxx	LLCTRL Equivalent to an idle packet Enables counting of slot 0 LLCTRL messages.
IDLE	b01000111	
PROTHDR	b1xxxxxxx	Protocol header Enables count of protocol headers in slot 0,1,2 (depending on slot umask bits)
NON_DATA	b10010111	Null flits transmitted to any slot

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.7 M2M Performance Monitoring

The M2M blocks manage the interface between the mesh (operating on both the mesh and the SMI3 protocol) and the memory controllers. The M2M acts as intermediary between the local CHA issuing memory transactions to its attached memory controller. Commands from the M2M to the MC are serialized by a scheduler and only one can cross the interface at a time.

2.7.1 M2M Performance Monitoring Overview

Each M2M box supports event monitoring through four 48b-wide counters (M2Mn_PCI_PMON_CTR/CTL{3:0}). Each of these four counters can be programmed to count almost any M2M event.

The M2M PMON also includes mask and match registers that allow a user to match packets of traffic heading to the DRAM or heading to the mesh, according to various standard packet fields such as message class, opcode, and so on.

2.7.2 M2M Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the M2M Box.

CLOCKTICKS

- **Title:**
- **Category:** Clockticks events

- **Event Code:** 0x01
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:**
- **Definition:** Shows legal flit time (hides impact of L0p and L0c).

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.8 M2PCIe* Performance Monitoring

M2PCIe* blocks manage the interface between the mesh and each IIO stack.

2.8.1 M2PCIe* Performance Monitoring Overview

Each M2PCIe* box supports event monitoring through four 48b-wide counters (M2n_M2PCIe*_PMON_CTR/CTL{3:0}). Each of these four counters can be programmed to count almost any M2PCIe* event.

The M2PCIe* counters can increment by a maximum of 5b per cycle. Only the counter 0 can be used for tracking occupancy events.

2.8.2 M2PCIe* Performance Monitoring Events

They are mesh stop events to track the ingress or egress traffic and mesh utilization (they are broken down by direction and ring type).

2.8.3 M2PCIe* Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the M2PCIe* Box.

CLOCKTICKS

- **Title:**
- **Category:** Clockticks events
- **Event Code:** 0x01
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:**
- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.9 M3 Intel® UPI Performance Monitoring

M3 Intel® UPI is the interface between the mesh and the Intel® UPI link layer. It is responsible for translating between the mesh protocol packets and the flits that are used for transmitting data across the Intel® UPI interface. It performs credit checking between the local Intel® UPI LL, the remote Intel® UPI LL and other agents on the local mesh.

The M3 Intel® UPI agent provides several functions:

- Interface between the mesh and Intel® UPI:
  - One of the primary attributes of the mesh is its ability to convey Intel® UPI semantics with no translation. For example, this architecture enables initiators to communicate with a local HA in exactly the same way as a remote HA on another 3rd Gen Intel® Xeon® Scalable Processor, previously codenamed Ice Lake socket. With this philosophy, the M3 Intel® UPI block is lean and does very little with regards to the Intel® UPI protocol aside from mirroring the request between the mesh and the Intel® UPI interface.
- Intel® UPI routing:
  - In order to optimize layout and latency, both full width Intel® UPI interfaces share the same mesh stop. Therefore, an Intel® UPI packet might be received on one interface and simply forwarded along on the other Intel® UPI interface. The M3 Intel® UPI has sufficient routing logic to determine if a request, snoop or response, is targeting the local socket or if it should be forwarded along to the other interface. This routing remains isolated to M3 Intel® UPI and does not impede traffic on the mesh.
- Intel® UPI home snoop protocol (with early snoop optimizations for DP):
  - The M3 Intel® UPI agent implements a latency-reducing optimization for dual sockets which issues snoops within the socket for incoming requests, as well as a latency-reducing optimization to return data satisfying Direct2Core (D2C) requests.

2.9.1 M3 Intel® UPI Performance Monitoring Overview

Each M3 Intel® UPI link supports event monitoring through three 48b-wide counters (M3_Ly_PCI_PMON_CTR/CTL{2:0}). Each of these three counters can be programmed to count almost any M3 Intel® UPI event. Only the counter 0 can be used for tracking occupancy events. Only the counter 2 can be used to count mesh events.

2.9.2 M3 Intel® UPI Performance Monitoring Events

They are mesh stop events to track the ingress or egress traffic and mesh utilization statistics (they are broken down by direction and ring type).

2.9.3 M3 Intel® UPI Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the M3 Intel® UPI box.

CLOCKTICKS

- **Title:**
- **Category:** Clockticks events
- **Event Code:** 0x01
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:**
- **Definition:**

Note: The event list available under NDA will become public and available for download closer to the product launch.

2.10 PCU Performance Monitoring

The PCU is the primary power controller for the Sapphire Rapids die, it is responsible for distributing power to core or uncore components and the thermal management. It runs in the firmware on an internal micro-controller and coordinates the socket power states.

The PCU algorithmically governs the P-state of the processor, the CPU Power State (C-state) of the core and the package C-state of the socket. It enables the core to go to a higher performance state "turbo mode" when the proper set of conditions are met. Conversely, the PCU throttles the processor to a lower performance state when a thermal violation occurs.

Through specific events, the OS and the PCU will either promote or demote the C-State of each core by altering the voltage and frequency. The System Power State (S-state) of all the sockets in the system is managed by the server legacy bridge in coordination with all socket PCUs.

The PCU communicates to all the other units through multiple PMLink interfaces on-die and message channel to access their registers. The OS and the BIOS communicate to the PCU through standardized MSR registers and ACPI.

Note: The power management is not completely centralized. Many units employ their own power saving features. Events that provide information about those features are captured in the PMON blocks of those units. For example, Intel® UPI link power saving states and memory CKE statistics are captured in the Intel® UPI PerfMon* and IMC PerfMon* respectively.

2.10.1 PCU Performance Monitoring Overview

The uncore PCU supports event monitoring through four 48-bit wide counters (PCU_MSR_PMON_CTR{3:0}). Each of these counters can be programmed (PCU_MSR_PMON_CTL{3:0}) to monitor any PCU event. The PCU counters can increment by a maximum of 5b per cycle.

Four extra 64-bit counters are provided by the PCU to track the P and C-State residence. Although documented in this manual for reference, these counters exist outside of the PMON infrastructure.

2.10.2 Additional PCU Performance Monitoring

2.10.2.1 PCU PMON Counter Control - Difference from Baseline

The following table defines the difference in the layout of the PCU performance monitor control registers from the baseline presented all across the previous chapter.

Figure 2-6. PCU Counter Control Register for Sapphire Rapids

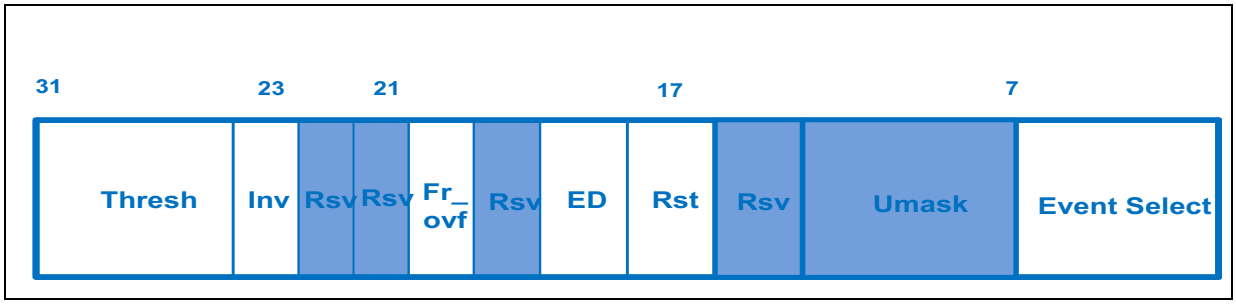


Table 2-17. Additional PCU Performance Monitoring Registers (MSR)

MSR Names Fixed (Non-PMON) Counters	Sizes (Bits)	Descriptions
PCU_MSR_CORE_P6_CTR	64	Fixed P-State residency counter
PCU_MSR_CORE_C6_CTR	64	Fixed C-State residency counter

Note: Address to the PCU specific filtering register can be found in Chapter 1. But there are some additional pieces of state of relevance to performance monitoring uses.

The PCU includes an extra MSR that tracks the number of reference cycles a core (any core) is in, C6 state. And the PCU also includes an extra MSR that tracks the number of reference cycles the package is in the C6 state. As mentioned before, these counters are not part of the PMON infrastructure so they cannot be frozen or reset, or otherwise controlled by the PCU PMON control registers.

Note: To be clear, these counters track the number of cycles the core is in (C6 state). It does not track the total number of cores in the C6 state in any cycle. For that, the user can see the regular PCU event list.

Table 2-18. PCU_MSR_CORE_{C6,P6}_CTR Register – Field Definitions

Fields	Bits	Attributes	HW Reset Values	Descriptions
event_count	63:0	RW-V	0	64-bit performance event counter

2.10.3 PCU Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the PCU Box.

CLOCKTICKS

- **Title:**
- **Category:** Clockticks events
- **Event Code:** 0x01
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:**

The event list available under NDA will become public and available for download closer to the product launch.

2.11 MDF Performance Monitoring

The MDF subsystem is a new IP built to support the new Intel® Xeon® architecture that bridges multiple dies with a embedded bridge system.

The MDF layers mesh protocol over the Embedded Multi-die Interconnect Bridge (EMIB).

Note: The EMIB is the physical layer and the MDF is the logical layer.

The MDF only exists in a subset of Sapphire Rapids SKUs.

2.11.1 MDF Performance Monitoring Overview

Each MDF box supports event monitoring through four 48b-wide counters (MDF_PMON_CTR/CTL{3:0}).

2.11.2 MDF Box Performance Monitor Event List

This section enumerates Intel® Xeon® Processors and Sapphire Rapids performance monitoring events for the MDF box.

CLOCKTICKS

- **Title:**
- **Category:** Clockticks events
- **Event Code:** 0x01
- **Maximum Inc/Cyc:** 1
- **Register Restrictions:**

Note: The event list available under NDA will become public and available for download closer to the product launch.