Apollo Lake Platform Intel Architecture Firmware Specification (Volume 1 of 2)

BIOS Specification

For Volume 2 of 2 refer Doc 559811

February 2017

Revision 1.2.2

Intel Confidential

Document Number: 559810

Contents

Figures

Tables

Examples

Revision History

Document Number	Revision Number	Description	Revision Date
559810	0.5.0	Initial Release.	September 2015
559810	0.6.0	• Added MSR 0x120, update MSR 0xE2 in Chapter 2. • Updated VTd requirement in Chapter 12. • Update W/A in Section 1.12. • Added Chapter 7.	November 2015
559810	0.6.1	• Updated – SMM debug update in Section 6.1.1.1. – Added 8254 CG enable/disable in Section 8.12.	December 2015
559810	0.7	• Section 2.19, WBINVALD_DL1 update • Section 3.4.1.3, MSR 0x120 usage update • Section 8.6.1.4.3 Cx ACPI mapping update • Section 8.9 update • Section 2.28 update for MONITOR Address for MWAIT W/A • Updated Section 7.1 Dram Configuration • Added Section 7.1.2 to describe the DQ mapping for PHY and So-Dimm	February 2016
559810	0.7.1	• Updated Section 1.6 based on POR. • Added Section 2.38 and 2.39 for MSR 0x614/0x615. • Updated Section 3.4 for cache usage. • Updated Section 8.5.1.1.1 • Updated IAFW SMIP Section.	March 2016
559810	0.8	• Updated all the chapters to align with SIC 0.8.0. • Removed Broxton related sections. • Removed MSR chapter which is maintained in EDS. • Added Section 7.12 • Added Appendix A	May 2016
559810	0.9	• Update Chapter 6 for offsets and default values of RMT/ODT related options.	June 2016
559810	1.0	• Added Section 6.1.2 for CA Mapping	July 2016
559810	1.1	• Updated Section 6 IA FW SMIP • Added SCC PG W/A Requirement	August 2016
559810	1.2	• Updated Cx Latency Table in Section 7.6.4.3	October 2016

		• Updated IA FW W/A List	
559810	1.2.1	• Updated Section 7.14.1 for BIOS_RESET_DONE • Updated IA FFW SMIP Chapter for ODT Setting • Added Note for TSEG size in Section 5.5	December 2016
559810	1.2.2	• Updated Section 1.8.3 for portID list • Updated Section 7.20.12 for LPC CLKRUN# requirement	December 2016

§

1 Introduction

This document assists Intel Architecture (IA) Firmware (FW) developers in supporting the Apollo Lake SoC (System-On-Chip).

The purpose of this document is to supplement the information provided in the External Design Specification (EDS) for use by BIOS vendors and Intel customers developing their own IA FW. Further detail of register descriptions for the Apollo Lake SoC can be found in the EDS.

1.1 Reference Documents

Document	Document No./Location
AP-485, Intel® Processor Identification and the CPUID instruction	http://www.intel.com/Assets/PDF/appnote/241618.pdf
Apollo Lake IA FW and BIOS Specification Volume 2 of 2	559811
Intel® 64 and IA-32 Architectures Software Developer's Manuals Volume 1: Basic Architecture Volume 2A: Instruction Set Reference, A-M Volume 2B: Instruction Set Reference, N-Z Volume 3A: System Programming Guide Volume 3B: System Programming Guide	253665 253666 253667 253668 253669 http://www.intel.com/products/processor/manuals/
Intel® Multiprocessor Specification, Version 1.4	http://www.intel.com/design/archives/processors/pro/docs/242016.htm
ACPI / Power Management	http://www.acpi.info/
ACPI / Power Management in Microsoft* operating systems	http://msdn.microsoft.com/en-us/windows/hardware/gg463220.aspx
System Management BIOS (SMBIOS) Specification	http://www.dmtf.org/standards/smbios
Apollo Lake External Design Specification (EDS)	557555 557556 557557
Serial Peripheral Interface Specification	Refer Note.
SD Specification	http://www.sdcard.org/
UFS Specification Version 2.0	http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs
eMMC Specification Version 5.0	http://www.jedec.org/

Document	Document No./Location
UEFI Specification 2.4	http://www.uefi.org/home/

Note: Contact local field representative for latest revision of this document.

1.2 Format and Notations

The target audience for this document is IA FW writers. The formats and notations used within this document model are those used by IA FW vendors. This section describes the formatting and the notations that will be followed in this document.

Number Format	Notation	Example
Decimal (default)	d	14d
Binary	b	1110b
Hex	h	0Eh
Data Type	**Notation**	**Size**
BIT	b	Smallest unit, 0 or 1
BYTE	B	8 bits
WORD	W	16 bits or 2 bytes
DWORD	DW	32 bits or 4 bytes
QWORD	QW	8 bytes or 4 words
Kilobyte	KB	1024 bytes
Megabit	Mb	1,048,576 bits or 128 KB
Megabyte	MB	1,048,576 bytes or 1024 KB
Gigabit	Gb	1,073,741,824 bits
Gigabyte	GB	1024 MB

This document will often refer individual bit fields within a register, as well as the registers themselves with their designated acronym, followed by the device, register address and bit field in parentheses. The reader is expected to be familiar with the register definitions for the CPU and devices attached, and must be capable of referencing the associated documentation listed in Reference Documents, if a more detailed description of the register field is required.

1.3 Errata

Refer appropriate processor specification update for workarounds for any errata that may be present on a particular processor stepping.

1.4 Register Definitions

Configuration registers for the APL SoC resides within the processor (called model specific registers or MSR's). The register set is accessible through software executed by the host processor. MSR registers are accessed as a Q- word (64-bit) quantity through the RDMSR and WRMSR instructions. Refer Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference M-Z for the description and use of these instructions. All multi-byte numeric fields use "little-endian" ordering (i.e., lower addresses contain the least significant parts of the field).Registers and register bits are assigned one or more of the following attributes. These attributes define the behavior of register and the bit(s) that are contained within. All bits are set to default values by warm reset. Sticky bits retain their states between warm resets.

Table 1-1. Model Specific Register (MSR) Access Attributes Nomenclature

RO	Read Only	If a register is read only, writes to this register have no effect.
WO	Write Only	If a register is write only, reads return undefined data
R/W	Read/Write	A register with this attribute can be read and written.
R/WC	Read/Write Clear	A register bit with this attribute can be read and written. However, a write of 1 clears (sets to 0) the corresponding bit and a write of 0 has no effect.
R/WL	Read/Write Lock	A register bit with this attribute can be read and written. However, if the register is locked by setting some other lock bit, the register bit is treated as read only.
R/WO	Read/Write Once	A register bit with this attribute can be read and written only once. Additional writes to R/WO bits will not alter their state.

Some of the registers described contain reserved bits. Software must deal correctly with fields that are reserved. On reads, software must use appropriate masks to extract the defined bits and not rely on reserved bits being any particular value. On writes, software must ensure that the values of reserved bit positions are preserved, unless otherwise noted. That is, the values of reserved bit positions must first be read, merged with the new values for other bit positions and then written back. The default state represents the minimum functionality feature set required to successfully bring up the system. Hence, it does not represent the optimal system configuration. It is the responsibility of the system initialization software (usually IA FW) to properly determine the cache configuration, operating parameters and optional system features that are applicable, and to program the registers accordingly.

Default values are in hexadecimal (h suffix) unless otherwise noted e.g., 0A5h. Binary values are denoted with a b suffix (e.g., 10100101b).

Table 1-2. Nomenclature Used for Register Default Values

X	Undefined	The default value is undefined.
S	Strap Value	The default value reflects an external hardware configuration strap setting or a fixed configuration value.
0-F	Hexadecimal value	The default value is a hexadecimal value.

1.5 Terminology

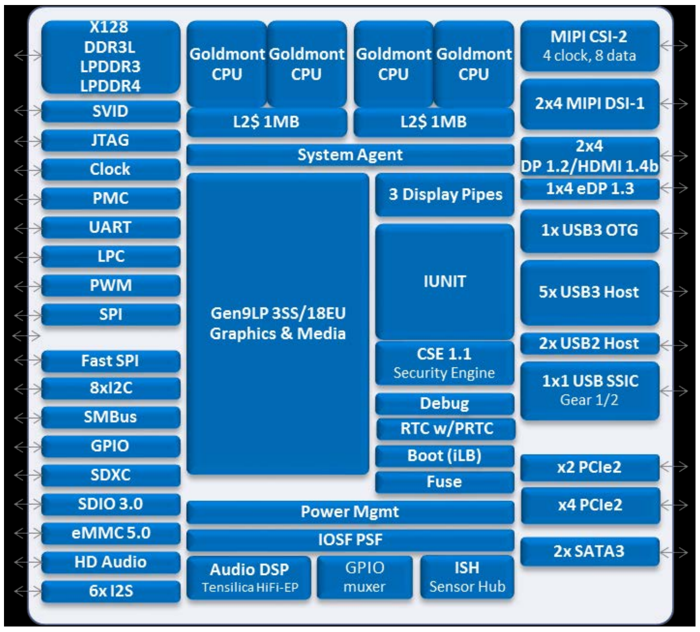
Term	Description
#	A '#' symbol after a signal name refers to an active low signal, indicating a signal is in the asserted state when driven to a low level.
AP	Application Processor. All the processor threads other than SBSP (System BSP).
BSP	Boot Strap Processor. Executes the IA FW's boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs.
Cache Line	Is defined as being 64-bytes in size, with each line having a unique set of MESI bits.
CMP	Core Multi-Processing (CMP) refers to a single physical package that utilizes multiple cores for multi-processing capabilities.
Core or CPU1 or CPU2	The silicon which contains 1 or more logical processors.
EC	Embedded Controller
Intel® 64	Intel® 64 Instruction Set Architecture. The instruction set architecture and programming environment of Intel's 64-bit processors which is a superset of and compatible with IA-32.
Intel® CTDP	Intel® Configurable TDP
Intel® DPPM	Intel® Dynamic Power Performance Management
Intel® DPTF	Intel® Dynamic Platform and Thermal Framework is envisioned as a platform level software framework that serves as a base for any power, thermal or other platform level control technology.
Intel® LPM	Intel® Low Power Mode
Logical Processor	The basic unit of processor hardware that allows the software executive in the operating system to dispatch a task or execute a thread context. Each logical processor can execute only one thread context at a time. A processor that may share execution resources with other processors in the same core.
Lx Cache	Used to denote cache hierarchy. L1 refers to the lowest level cache, separated for instruction and data L2 refers to the unified second level cache
MSR	Model Specific Register (MSR) as the name implies is model specific and may change from processor model number (n) to processor model number (n+1). An MSR is accessed by setting ECX to the register number and executing either the RDMSR or WRMSR instruction. The RDMSR instruction will place the 64 bits of the MSR in the EDX: EAX register pair. The WRMSR writes the contents of the EDX: EAX register

Term	Description
	pair into the MSR.
Physical Processor/Thread	A package which contains 1 or more cores that share a common connection to the system bus.
Package	Refers to all logical processors within a single physical processor package and is equivalent to a physical processor.
PAM	Programmable Attribute Map
RANK	One side of a double sided DIMM. Sometimes also referred to as Row.
SPI	Serial Peripheral Interface
Shared	"Shared" as used in this document to refer the general purpose registers, the Model Specific Registers (MSR), and the functional units that are common to each logical processor within the same processor core.
Unique	"Unique" as used in this document to refer the general purpose registers, the Model Specific Registers (MSR), and the functional units that are unique to each logical processor.
VID	Voltage Identification (VID) is a binary pattern output from the processor that tells the voltage regulator the voltage required to operate the processor.

1.6 Apollo Lake SoC Features – Summary

Apollo Lake system block diagrams are shown below. Some features are limited to certain SKUs. For a complete list of features, refer latest Apollo Lake SoC EDS.

Figure 1-1. Apollo Lake System Block Diagram



- Up to four IA-Compatible Low Power Intel Atom™ processor cores.
- SoC Transaction Router/System Agent.
- Memory Controller supports dual-channel DDR3L-RS/LPDRR3, LPDDR4 and WIO2, ECC, up to 2 ranks per channel (4 ranks in total) and 2x64 LPDDR3 DRAM device data width and soldered down DRAM devices.
- Intel's 9th generation (Gen 9) graphics and media encode/decode engine. Supports 3D rendering, media compositing and video encoding, Graphics Burst enabled through energy counters, DX 11+, EGL 1.4, OGL ES 3.0, OpenCL 2.x. Full HW acceleration for decode of HEVC (H.265), H.264 @5.1, SVC, VP8 1080p, VP8 4K, VP8 stereo, MPEG4 ASP/SP, AVS, BD2.4, H.263 and VP9. Full HW acceleration for encode of HEVC (H.265), H.264@ 4.2, VP8 1080p, VP8 4K, VP8 stereo, SVC, AVS, H.263 and VP9.
- Display controller with three pipes. It supports 2 MIPI DSI ports with Stereoscopic 3D formats, 3 DDI ports to enable eDP 1.3, DP 1.2, DVI, or HDMI 1.4b, 2 panel power sequence for 2 eDP ports, Audio on DP and HDMI, and Intel Display Power Saving Technology (DPST) 6.0, Panel Self Refresh (PSR) and Display Refresh Rate Switching Technology (DRRS).
- One Image Signal Processor supports up to three MIPI CSI port, support up to 24MP sensors and stereoscopic video.
- Low Power Engine (LPE) Audio controller, a complete audio solution based on an internal audio processing engine which includes several I2S-based output ports.
- Peripheral IO's
    - ISH
    - eMMC
    - SD Card
    - SDIO
    - USB3 – 6 SS ports, 2 HS ports
    - USB3 OTG
    - Intel® Trusted Execution Engine (Intel® TXE)
    - PCIe
    - SATA
    - Low Power Input Output:
        - 2x DMA
        - 11x I2C
        - 2x I2S
        - 2x PWM
        - 3x HS-UART
        - 5x SPI
        - 1x SMBUS 2.0
    - Platform Controller Unit (PCU):
        - Fast SPI
        - UART
        - GPIO
        - Legacy Block: LPC, RTC (support discrete VR's when there is no PMIC), iLB, and so on.

1.7 Required IA FW Modifications

The following is a list of the features requiring IA FW support that are deemed essential for the Apollo Lake SoC.

- Correct processor identification.
- Correct processor speed identification.
- L1 and L2 cache enabling, including proper initialization of the CR0.CD bit.
- Correct Memory Type Range Register (MTRR) configuration.
- Correct microcode update loading on all processor cores and logical processors in the system.
- Management of multiple microcode updates in the flash ROM.
- SMI / SMRAM / SMRR /PRMRR initialization.
- Construction of ACPI tables for ACPI operating systems.
- Support for Core Multi-Processing (CMP).
- Intel Thermal Monitor initialization.
- Enhanced Intel SpeedStep® Technology initialization.
- APIC interrupt model initialization and reporting.
- Appropriate enabling of low power features.
- Initialize the Machine-Check Architecture for each logical processor in the system.
- Clear all bits in IA32_MCi_STATUS registers during a cold boot. IA FW must not clear the Machine check status bits on a warm boot.
- Ensure all processors have identical configuration (same cache sizes). Force a shutdown if the configurations are dissimilar.
- Each logical processor at the end of the MP init sequence must test the microcode update revision again and ensure the microcode update is fully loaded on every thread. The microcode update only needs to be reloaded if the revision of the microcode update in the IA FW is greater than what is installed in the logical processor.
- Program the IA FW_RESET_CPL register after microcode patch and all PM relevant configurations.
- Using the mailbox, the IA FW must write the IA core Processor Signature (CPUID.(EAX=1):EAX[31:0]) to the PECI DATA space (offset 0h).
- Using the mailbox, the IA FW must write the platform ID (MSR 017h[52:50]) to the PECI DATA space (offset 1h).
- Using the mailbox, the IA FW must write the Microcode Update Revision (MSR 08Bh[63:32]) to the PECI DATA space (offset 4h).
- Replace all SW HASH algorithm with SHA instructions to reduce space and speed up SHA operations.
- At the end of POST and before control is passed to the operating system (OS), the IA FW must set the POWER_MISC [Enable_IA_Untrusted_Mode] bit for each core/ thread (MSR 0x120[bit 6]).

1.8 Apollo Lake SoC IOSF-SB Bus Access

Apollo Lake SoC contains a Message Network called IOSF Sideband (IOSF-SB, SB, or sideband) as an internal communication medium for simple, low bandwidth, sideband communication. Host software (IA FW) can access the message network via configuration registers in device 13 to access private (i.e. non host-mapped) registers and to initiate messages (for example, interrupts, or power management events).

Access to this network by the CPU is needed for only a handful of SoC operations including but not limited to:

- Configuring system DRAM and Uncore Units.
- Setting CPU configuration and causing a CPU-Only reset.
- SoC Performance Monitoring (PMON).
- Thermal management setup.
- Display IO and PLL configuration.
- GPIO configuration.

Note: Some SoC registers are ONLY accessible to software via this method.

1.8.1 IOSF-SB Register Addressability

Access to the message bus space is through the SoC Transaction Router's PCI configuration registers. This unit relies on three 32-bit PCI configuration registers to generate messages:

IOSF-SB MMIO ADDR: PCI[B:0,D:13,F:0] | 0x10

1.8.2 IOSF-SB Access Mechanisms

P2SB MMIO Register Interface

Addr[63:24]	Addr[23:16]	Addr[15:2]	Addr[1:0]
BAR	Target Port ID	Register offset	2'b00

1.8.3 Apollo Lake SoC IOSF-SB Port Summary

Port ID	Function
0x30	Gen
0x32	Iunit
0x46	DPTF
0x82	PMC

Port ID	Function
0x88	NPK
0x90	LPSS
0x92	HAD
0x93	SPI
0x94	ISH
0x95	PMC IOSF2OCP
0x97	TXE
0x98	ISH
0xA2	xHCI Host controller
0xA4	xDCI device controller
0xA5	USB3 MODPHY
0xA7	USB2 PHY
0xA8	EXI
0xB3	PCIE0
0xB4	PCIE1
0xC0	GPIO Southwest Community
0xC4	GPIO Northwest Community
0xC5	GPIO North Community
0xC7	GPIO West Community
0xCD	SMBus
0xD0	ITSS
0xD1	RTC
0xD2	LPC
0xD4	P2SB
0xD6	SCC

1.9 Hiding SoC Internal Devices

APL SoC provides a mechanism of hiding the PCI Configuration space of the internal PCI devices. Devices that are not used in the system can be hidden before the IA FW loads the OS to prevent the OS from erroneously loading drivers. Devices should be hidden before the IA FW assigns system resources to prevent unused and unnecessary holes from being created in the system address space. Devices can be hidden / restored by setting / clearing the proper bit in PBASE+0x34 or PBASE+0x38.

When hiding (disabling) a function, only the configuration space is disabled by internal logic. Software must ensure that all functionality within the function (such as memory spaces, I/O spaces, and DMA engines) is disabled and PCI command register 04h is cleared prior to disabling the function.

The PCI specification requires that a multi-function device must implement function 0. As a general rule, IA FW must ensure that if function 0 of a multi-function device is disabled (hidden) via the APL SoC Function Disable register, then all the other functions of the same device should be disabled as well in order to be compliant with PCI spec. For example, when IA FW disables D28:F0 (root port #1), it should disable all other functions of device 28 as well.

Table 1-3. PCI Device/Function Disable Registers

Device / Function	Device Name	Register Location
Device 0 / Function 0	SoC Transaction Router	N/A
Device 0 / Function 1	DPTF	N/A
Device 0 / Function 2	NPK	N/A
Device 0 / Function 3	Gaussian Mixture Models	PBASE+0x34[31]
Device 2 / Function 0	Graphics and Display Port	N/A
Device 3 / Function 0	Iunit	N/A
Device 13 / Function 0	P2SB	N/A
Device 13 / Function 1	PMC	N/A
Device 13 / Function 2	SPI	PBASE+0x34[30]
Device 13 / Function 3	SSRAM	PBASE+0x34[29]
Device 14 / Function 0	Audio	PBASE+0x34[28]
Device 15 / Function 0	HECI1	PBASE+0x34[27]
Device 15 / Function 1	HECI2	PBASE+0x34[26]
Device 15 / Function 2	HECI3	PBASE+0x34[25]
Device 16 / Function 0	HOFFL	
Device 17 / Function 0	ISH	PBASE+0x34[24]
Device 18 / Function 0	SATA	
Device 19 / Function 0	PCIe-A0	
Device 19 / Function 1	PCIe-A1	
Device 19 / Function 2	PCIe-A2	

Device / Function	Device Name	Register Location
Device 19 / Function 3	PCIe-A3	
Device 20 / Function 0	PCIe-B0	
Device 20 / Function 1	PCIe-B1	
Device 20 / Function 0	MEX 0	PBASE+0x34[23]
Device 20 / Function 1	MEX 1	PBASE+0x34[22]
Device 21 / Function 0	USB-Host	PBASE+0x34[1]
Device 21 / Function 1	USB-Device	PBASE+0x34[0]
Device 22 / Function 0	I2C0	PBASE+0x34[21]
Device 22 / Function 1	I2C1	PBASE+0x34[20]
Device 22 / Function 2	I2C2	PBASE+0x34 [19]
Device 22 / Function 3	I2C3	PBASE+0x34 [18]
Device 22 / Function 6	I2C 6/7	PBASE+0x34 [18]
Device 22 / Function 7	PMC	PBASE+0x34 [18]
Device 23 / Function 0	I2C4	PBASE+0x34 [17]
Device 23 / Function 1	I2C5	PBASE+0x34 [16]
Device 23 / Function 2	I2C6	PBASE+0x34 [15]
Device 23 / Function 3	I2C7	PBASE+0x34 [14]
Device 24 / Function 0	UART0	PBASE+0x34 [13]
Device 24 / Function 1	UART1	PBASE+0x34 [12]
Device 24 / Function 2	UART2	PBASE+0x34 [11]
Device 24 / Function 3	UART3	PBASE+0x34 [10]
Device 25 / Function 0	SPI0	PBASE+0x34 [9]
Device 25 / Function 1	SPI1	PBASE+0x34 [8]
Device 25 / Function 2	SPI2	PBASE+0x34 [7]
Device 26 / Function 0	PWM	PBASE+0x34 [6]
Device 27 / Function 0	SD Card	PBASE+0x34 [5]
Device 28 / Function 0	eMMC	PBASE+0x34 [3]
Device 29 / Function 0	UFS	PBASE+0x34 [2]
Device 30 / Function 0	SDIO	PBASE+0x34 [4]
Device 31 / Function 0	LPC	
Device 31 / Function 1	SMBUS	

1.10 PCAT or IASOC (Hardware-Reduced ACPI)

1.10.1 PCAT and IASOC in Existing Products

Version 5.0 of “The Advanced Configuration and Power Interface Specification” introduces the concept of “Hardware-Reduced ACPI”. This is a class of computer where the “ACPI Fixed Hardware interface is not implemented, and software alternatives for many of the features it supports are used instead”. The term “IASOC” refers to a system where the main CPU is x86 based, and the system is exposed to the OS as “Hardware-Reduced ACPI”.

Conversely, the term PCAT is used here to refer to systems that implement the ACPI Fixed Hardware interface, in line with version 5.0 and earlier versions of the ACPI spec.

Prior to APL, for “Bay Trail-T”, only IASOC is supported. For all other Intel x86-based platforms, only PCAT is supported.

1.10.2 PCAT and IASOC

APL is designed to be suitable for both traditional PC products, and also recent products such as tablets. It can be configured and exposed to the OS as either a PCAT or IASOC system. For any particular product, the designers and IA FW writers should carefully consider which mode is most suitable for the product, and whether to offer an IA FW-configurable choice.

IA FW indicates the system type to the OS using the “HW_REDUCED_ACPI” flag of the FADT table.

In PCAT type systems, IA FW will clear this flag, and should prepare hardware resources such as SCI interrupt mechanisms according to the full ACPI hardware specification.

For IASOC systems, IA FW should set this flag, and assign alternate mechanisms such as GPIO in their place. Likewise, the platform resourced described to the OS in FADT and similar tables will change according to the system type chosen.

The table below covers some of the key factors to be used when determining whether the APL-based system is to be configured as PCAT or IASOC (Hardware-Reduced ACPI). For further detail, consult the ACPI specification.

Table 1-4. Key Factors in Selecting PCAT or IASOC Configuration

Feature	PCAT implementation	IASOC implementation
Typical system	Desktop, Notebook, high-end tablet	Tablet and similar systems that do not require PCAT-type features
ACPI hardware model	ACPI Fixed Hardware interface	Hardware-reduced ACPI
IA FW type supported	UEFI or Legacy EFI	UEFI only
OS type supported	ACPI OS (version 5.0 and prior versions) or Non-ACPI OS	ACPI 5.0 and later compliant OS

Feature	PCAT implementation	IASOC implementation
Relationship to 64-bit OS	PCAT systems will generally be used with a 64-bit OS	Technically possible but no known use case
Relationship to 32-bit OS	Diminishing use case	Typical use case
PMBLK, GPE, 8254 timer, 8259 interrupt controller, SCI interrupts	Available during OS boot and OS operation	Not available
Limitations regarding PCI/ACPI-enumerated devices	ACPI-enumerated devices may not be discovered by an OS that does not support ACPI version 5.0	Power Management Events (PME) of PCI-enumerated devices require special handling due to the lack of PMBLK
Relationship to “Low power S0 Idle Capable” of ACPI 5.0 specification	Anticipation of systems both with and without “Low power S0 Idle Capable”	Anticipation that most IASOC systems will run as “Low power S0 Idle Capable”
Supported platform event delivery to OS	SCI and/or Dedicated IOxAPIC IRQ	Dedicated IOxAPIC IRQ and/or Shared GPIO IRQ
S3/S4/S5 sleep state OS support	Proven	No S3 support. S4 and S5 supported through OS call to UEFI.

1.11 R/WO Registers

Some of the registers in APL SoC are implemented as Read/Write-Once (RWO) while some of them are defined as Read-Only by the PCI Express* specification. It is required that IA FW must write to such registers once during POST and before booting to OS, so that the registers will become Read-Only.

The table below lists some of known registers in APL. Refer EDS for complete register details.

Table 1-5. RWO Registers

Dev:Func	Register Offset	Register Name	Bits
D2:F0	0x2C	Subsystem ID	31:0
D12:F0	0x2C	Subsystem ID	31:0
D13:F0˜F3	0x2C	Subsystem ID	31:0
D14:F0	0x2C	Subsystem ID	31:0
D15:F0~F2	0x2C	Subsystem ID	31:0
D16:F0	0x2C	Subsystem ID	31:0
D17:F0	0x2C	Subsystem ID	31:0
D20:F0~F1	0x2C	Subsystem ID	31:0
D21:F0~F1	0x2C	Subsystem ID	31:0
D22:F0~F3	0x2C	Subsystem ID	31:0

Dev:Func	Register Offset	Register Name	Bits
D23:F0~F3	0x2C	Subsystem ID	31:0
D24:F0~F3	0x2C	Subsystem ID	31:0
D25:F0~F2	0x2C	Subsystem ID	31:0
D26:F0	0x2C	Subsystem ID	31:0
D27:F0	0x2C	Subsystem ID	31:0
D28:F0	0x2C	Subsystem ID	31:0
D29:F0	0x2C	Subsystem ID	31:0
D30:F0	0x2C	Subsystem ID	31:0

§

2 Cache Control

Caching in Intel 64 and IA-32 architecture processors is controlled by the CR0.CD bit, Memory Type Range Registers (MTRR), System Management Range Register (SMRR) and Page Attribute Table (PAT). It is the responsibility of the IA FW to configure the memory map and cache ability identically on all processors to ensure cache coherency.

The PAT is typically configured by the OS. Therefore, the PAT is not discussed in this document. For details about the PAT, refer Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.

The first level caches in the APL SoC are referred to as the "L1 Instruction" cache (L1I) and "L1 Data" cache (L1D). The L1I and L1D sizes should be determined by executing CPUID.(EAX=4). Each core in the APL SoC has a unique L1I and L1D. The threads within a core share that core's L1I and L1D.

The second level cache in the APL SoC may be referred to by several names, the "L2" (L2) and "Mid-Level Cache" (MLC). The MLC size should be determined by executing CPUID.(EAX=4).

2.1 MTRR Initialization and Cache Enable Requirements

The IA FW must program the Memory Type Range Registers (MTRRs) as well as each logical processor's CR0.CD bit to enable caching. All MTRR's must be programmed with the same memory type and ranges for all logical processors in a platform.

The APL SoC does not support individual cache level enabling. Caches for a core are either all enabled or all disabled. When the MTRRs are configured and enabled and the CR0.CD bits from each thread are cleared to 0, the caches for that core are enabled.

Note: Three bits determine the enabled or disabled state of the caches in a core on the APL SoC (Core CR0.CD bit and the IA32_MTRR_DEF_TYPE.E bit). To enable caching the CR0.CD bit on each core must be cleared to 0. The CR0.CD bit must be configured to be identical on all cores - otherwise the processor is in an unsupported mode of operation. It is expected that the state of CR0.CD for various threads will differ for a short time during the MP initialization sequence.

Note: The IA FW must specify in the MTRRs which address ranges are cacheable. Alternatively, if enough memory exists to surpass the maximum cacheability range of the processor, the IA FW may configure the MTRR_DEF_TYPE.Type (Bits 7:0) as WB and use the variable MTRRs to create UC memory holes in the system memory map. Refer Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 for details on programming the MTRRs and CR0.

Note: It is also important to note that MTRR's are unique for each processor core. As with any multi-processor configuration, it is software's responsibility to ensure that the MTRR's for all logical processors in the system are configured with the same ranges and memory types. Failure to adhere to this requirement will result in the processor running out of specification.

2.2 Disabling Caches

The L1 and L2 caches are enabled by the IA FW during POST. It is recommended that the caches be left enabled. It is recommended that the L1 and L2 caches be left enabled, however, if the IA FW implements a cache disable feature, the following algorithm is recommended. This algorithm does not modify the variable range and fixed range MTRRs.

The IA FW should execute the CPUID instruction with EAX = 2 to get the cache descriptors. If the value 40h is returned as one of the cache descriptors, then no L2 exist. When 40h is not returned as a cache descriptor, the processor has an L2 cache.

Example 2-1. Disable Caches

```
Disable_Caches()
{
Suppress_all_IO_traffic()
K = 0
WHILE (K < num_logical_processors)
{
// Obtain current value of K. This forces both temp and K
// into the L1 cache. Note temp could also be maintained
// in a general-purpose register.
temp = K
//Wait_until_all_processors_sign_in_at_barrier()

IF (logical_processor_id == K)
       {
  wait_10_msecs_delay_loop()
// This time delay is required!
// In the worst case, this allows the barrier semaphore
// to settle to a shared state
// Step 1. Enter the cache disabled mode.
       CR0.CD    1
// Step 2. Flush caches using the WBINVD instruction.
// Note that WBINVD will create exclusive access for
// the logical processor within a package, thus other
// logical processors within the package will not cause
// re-caching of the line during the WBINVD, or
// coincidental snoop generation. The caches must be
// flushed when CR0.CD == 1 to ensure system memory
// coherency. If the caches are not flushed now,
// cache hits on reads will still occur and data
// will be read from valid cache lines.
WBINVD

// Step 3. Disable the MTRRs and set the
// default memory type to uncached or set all
// MTRRs for the uncached memory type.
// For more information, see the discussion of the TYPE
// field and the E flag in Section 9.12.2.1 "MTRRdefType
// Register" of the "IA32 Software Developers Manual,
// volume 3: System Programming Guide"
       MTRRdefType[11]   0

       WBINVD

// Step 4. Re-enable the MTRRs
  MTRRdefType[11]       1
       CR0.CD     0
  K++
       }
ELSE
WHILE (temp == K)
}
}
```

This algorithm prevents bus snoop traffic from the other processors, which would otherwise cause the initializing processor to hang. The temp and K variables must be cached in L1 by each processor. Also the Memory Type Range Register (MTRR) for the data segment must be set to WB (write-back) memory type.

2.3 Re-Enabling Caches

If the IA FW implements a cache disable option, it must also implement a cache re-enable algorithm. The following algorithm is recommended for re-enabling the caches. This algorithm assumes that variable range and fixed range MTRRs have not been modified.

Example 2-2. ReEnable Caches

```
ReEnable_Caches()
{
  Suppress_all_IO_traffic()
      K ← 0
  WHILE (K < num_logical_processors)
      {
// Obtain current value of K. This forces both temp and K
// into the L1 cache Note that temp could also be
// maintained in a general-purpose register.
      temp ← K
      Wait_until_all_processors_sign_in_at_barrier()

    IF (logical_processor_id == K)
        {
```

```
    wait_10_msecs_delay_loop()
// This time delay is required
// In the worst case, this allows the barrier semaphore
// to settle to a shared state.
// Step 1. Enter the no-fill cache mode.
        CR0.CD ← 1
    CR0.NW ← 0

// Step 2. Flush caches using the WBINVD instruction.
// Note that WBINVD will create exclusive access for the
// logical processor within a package, other logical
// processors within the package will not cause
// re-caching of the line during the WBINVD, or
// coincidental snoop generation. The caches must be
// flushed when CR0.CD == 1 to insure system memory
// coherency. If the caches are not flushed now,
// cache hits on reads will still occur and
// data will be read from valid cache lines.
    WBINVD

// Step 3. Disable the MTRRs and set the default
// memory type to uncached or set all MTRRs for the
// uncached memory type.  For more information, see
// the discussion of the TYPE field and the E flag in
// Section 9.12.2.1 ''MTRRdefType Register''
// of the ''IA32 Software Developers Manual
// volume 3: System Programming Guide''
        MTRRdefType[11] ← 0
    WBINVD

// Step 4. Re-enable the MTRRs
        MTRRdefType[11] ← 1

// Step 5. Enter the cache enabled mode (normal mode)
    CR0.CD ← 0
        CR0.NW ← 0
    K++
        }
  ELSE
       WHILE (temp == K)

       }
}
```

2.4 Using Cache for Stack and Code during Memory Initialization

APL SoC allows for using the processor cache as both code and stack prior to system memory availability. The limitations and restrictions must be followed explicitly. This is not a general purpose configuration and may only be used as documented.

When properly configured, the IA FW may enable caching a small amount of code and a small amount of stack/data on one logical processor (the System BSP).

Using this configuration, it is reasonable to write high-performance IA FW code utilizing a read/write data stack. Conventional BIOS has been effectively limited to stack-less assembly code. Recent processors have allowed the cache to be configured to provide a small amount of read/write data stack. With stack capability, it is reasonable to write pre-memory code in higher level languages, like C, and with access to significantly more volatile data. The APL SoC allows a further enhancement in that the IA FW may also contain code while using the cache for stack. This can significantly improve the execution speed for pre-memory initialization code.

It is required that IA FW utilizes this configuration option in order to support Intel's memory reference code package. However, conventional BIOS that executes stack-less assembly code may not directly benefit from utilizing this configuration. IA FW that has been updated to utilize the cache for data stack will likely see a significant performance improvement when utilizing this configuration.

The use of the processor cache for stack and code must be discontinued as soon as possible after memory initialization.

2.4.1 Enabling Cache for Stack and Code Use Prior to Memory Initialization

After power-on reset and prior to performing the memory detection, the IA FW will configure the whole L2 cache as WB type. The L2 cache could then be used as both stack and code.

The following steps must be executed by the IA FW to place the BSP in this mode and correctly configure the cache for stack and code use:

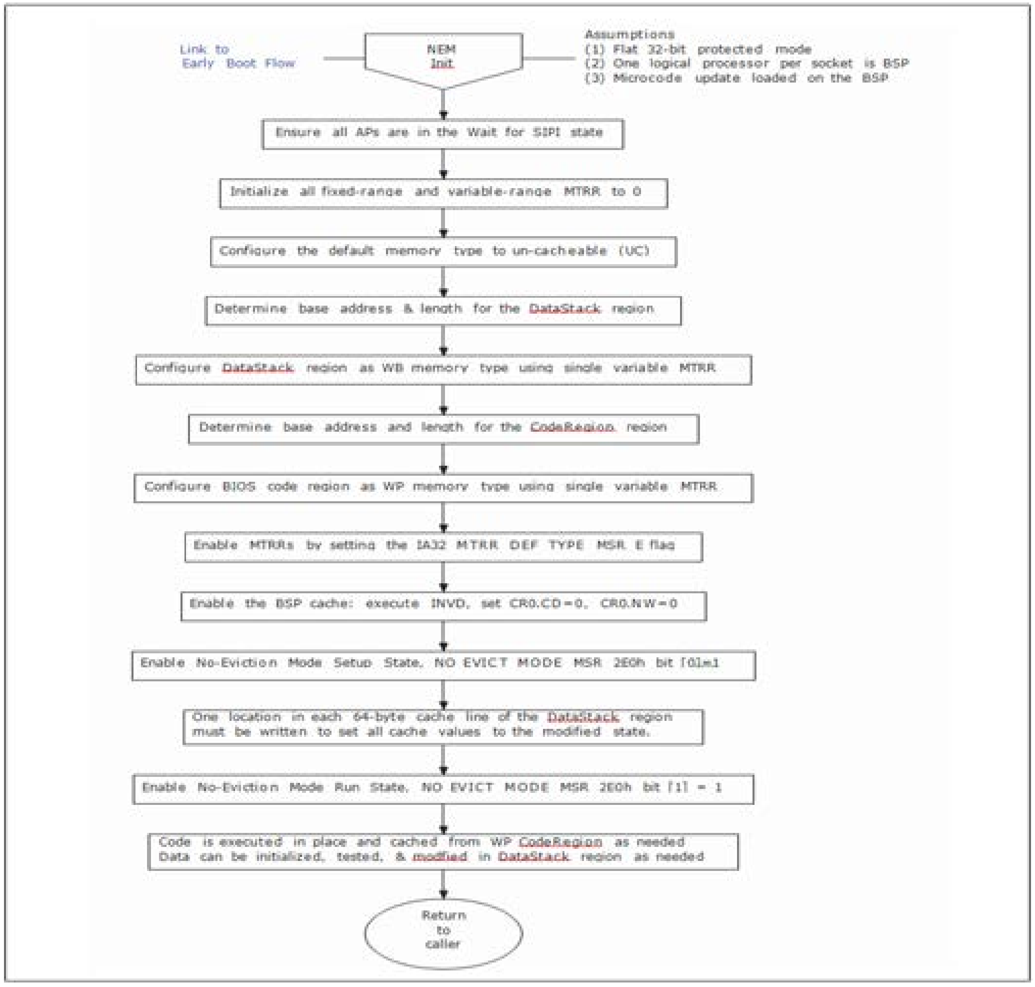
1. Ensure that the system is in flat 32-bit protected mode.
2. Ensure that only one logical processor in the system is the BSP (Required step for clustered systems).
3. Ensure all APs are in the Wait for SIPI (Startup Inter Processor Interrupt) state. If any APs are awake, the IA FW must put it back into the Wait for SIPI state by issuing a broadcast INIT IPI to all excluding self.
4. Ensure that all variable-range MTRR valid flags are clear and IA32_MTRR_DEF_TYPE
5. MSR E flag is clear. This is the default state after hardware reset.
6. Initialize all fixed-range and variable-range MTRR register fields to 0.
7. Configure the default memory type to un-cacheable (UC) in the IA32_MTRR_DEF_TYPE MSR.
8. Determine the base address and length for the CAR region.
9. Verify the base address and length for the CAR region.

10. Configure the CAR region as write-back (WB) cacheable memory type using the variable range MTRRs.
11. Enable the MTRRs by setting the IA32_MTRR_DEF_TYPE MSR E flag.
12. Enable the L2 cache by setting bit 8 of the BBL_CR_CTL3.
13. Enable the logical processor's (BSP) cache: execute INVD and set CR0.CD = 0, CR0.NW = 0.
14. Enable No-Eviction Mode Setup State by setting IA32_BIOS_CACHE_AS_RAM MSR 2E0h bit [0] = '1'
15. Modify all values in the CAR region by writing to each location.

Note: All values must be modified to set the cache values to the modified state.

16. Enable No-Eviction Mode Run State by setting IA32_BIOS_CACHE_AS_RAM MSR 2E0h bit [1] = '1'.

Figure 2-1. No Eviction Mode Initialization Flow

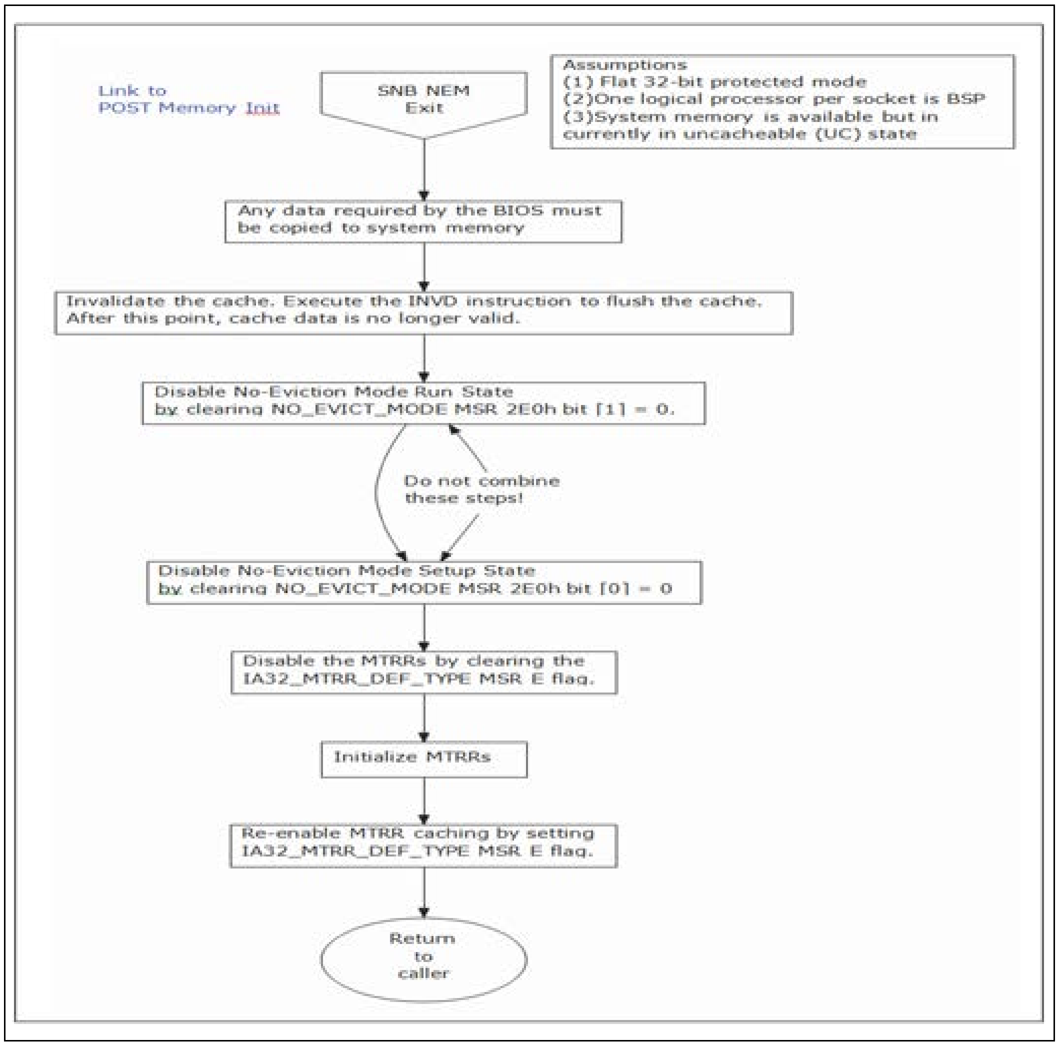

Link to Early Boot Flow
NEM Init
Assumptions
(1) Flat 32-bit protected mode
(2) One logical processor per socket is BSP
(3) Microcode update loaded on the BSP
Ensure all APs are in the Wait for SIPI state
Initialize all fixed-range and variable-range MTRR to 0
Configure the default memory type to un-cacheable (UC)
Determine base address & length for the DataStack region
Configure DataStack region as WB memory type using single variable MTRR
Determine base address and length for the CodeRegion region
Configure BIOS code region as WP memory type using single variable MTRR
Enable MTRRs by setting the IA32 MTRR DEF TYPE MSR E flag
Enable the BSP cache: execute INVD, set CR0.CD=0, CR0.NW=0
Enable No-Eviction Mode Setup State, NO EVICT MODE MSR 2E0h bit [0]=1
One location in each 64-byte cache line of the DataStack region must be written to set all cache values to the modified state.
Enable No-Eviction Mode Run State, NO EVICT MODE MSR 2E0h bit [1] = 1
Code is executed in place and cached from WP CodeRegion as needed Data can be initialized, tested, & modified in DataStack region as needed
Return to caller

2.4.2 Disabling Cache for Stack and Code Use after Memory Initialization

As soon as possible after configuring initial system memory, the IA FW should transit to normal cache mode. Below are the steps required to transition to normal cache mode.

- Any stack data required by the IA FW must be copied to the initial system memory.
- The initial system memory must be in the un-cacheable (UC) state.
- Invalidate the cache. Execute the INVD instruction to flush the cache. After this point, cache data is no longer valid and the next read will come from the flash. IA FW must ensure no data modification occurs until No-Eviction mode exit is completed.
- Disable the MTRRs by clearing the IA32_MTRR_DEF_TYPE MSR E flag.
- Disable No-Eviction Mode Run State by clearing IA32_BIOS_CACHE_AS_RAM MSR 2E0h bit [1] = 0.
- Disable No-Eviction Mode Setup State by clearing IA32_BIOS_CACHE_AS_RAM MSR 2E0h bit [0] = 0.
- Do not combine steps 4 and 5.
- Configure system memory and cache as specified in the Intel® 64 and IA-32
- Architectures Software Developer's Manuals, Volume 3A/3B: System Programming
- Guide. For example, initialize any remaining memory and MTRRs.
- Continue with POST.

Figure 2-2. No Eviction Mode Exit Flow


Link to
POST Memory Init
SNB NEM
Exit
Assumptions
(1) Flat 32-bit protected mode
(2)One logical processor per socket is BSP
(3)System memory is available but in currently in uncacheable (UC) state
Any data required by the BIOS must be copied to system memory
Invalidate the cache. Execute the INVD instruction to flush the cache. After this point, cache data is no longer valid.
Disable No-Eviction Mode Run State by clearing NO_EVICT_MODE MSR 2E0h bit [1] = 0.
Do not combine these steps!
Disable No-Eviction Mode Setup State by clearing NO_EVICT_MODE MSR 2E0h bit [0] = 0
Disable the MTRRs by clearing the IA32_MTRR_DEF_TYPE MSR E flag.
Initialize MTRRs
Re-enable MTRR caching by setting IA32_MTRR_DEF_TYPE MSR E flag.
Return
to
caller

2.4.3 Restrictions in Using Cache for Stack and Code

There are a number of specific restrictions that are relevant to configuring and utilizing the cache in No Eviction Mode.

- The BSP must be in flat 32-bit protected mode. Other modes are not validated or supported. Refer the Intel® 64 and IA-32 Architectures Software Developer's Manuals, Volume 3A/3B: System Programming Guide for details on enabling protected mode.
- All AP must be in the Wait for SIPI state.
- The CAR region used for stack should be kept xxx cache lines away from the area used for code. This is to ensure the modification of the data area would not result in the flush of the code area.
- The CAR region should be configured using only one variable MTRR register.
- The CAR size is determined by the L2 size.

- The CAR region must be in the range from (4GB-64MB) to 4GB.
- MLC and LLC are enabled at power on reset.
- Cache must be expanded to all ways. No power mode enabling is allowed with this configuration. This is the default state at reset.
- System must be in placed in Normal Cache Mode (CR0.CD = 0, CR0.NW = 0) after configuration of CAR.
- CAR must be modified after Normal Cache Mode to set all lines to modified (M) cache state and before use to store data.
- Interrupts must be disabled.
- Paging is not allowed.
- Code must be exception free.
- Code and platform design must guarantee no snooping of the memory in use.
- Code must ensure that the data in the CAR region is not evicted during initialization of the system memory.
- This explicit flush should be done only when code is copied into the CAR. For usage of CAR where the code is filled by fetching as opposed to copying into CAR this flushing is not required. With the TXE assisted boot IBBL copies the contents of 'FIT Table/IBB binary' from TXE SRAM to x86 CPU Cache As RAM (CAR). IBBL has to ensure each chunk of data that's copied from TXE SRAM to CAR via TXE ring buffer protocol is flushed to L2 cache from the L1 data cache by writing Bit8 of MSR 0x120 before executing that code. Not setting this MSR bit before executing the code copied into CAR may cause the lines to be evicted from the L1 data cache leading to corruption when the CAR is not backed up by real memory. Invoking this flush when not in CAR mode could lead to undefined behavior.

Note: Writing to MSR 0x120 bit8 is a command bit which initiates L1Data flush. Once written, it initiates the action (in this case flush) and no need to clear it.

2.4.4 Notes for Using Cache for Stack and Code

There are a few important considerations to take into account when deciding if use is appropriate:

- No special hardware protection is provided to suppress evictions due to snooping.
- Use of INVD will result in loss of DataStack data.
- Use of WBINVD or CLFLUSH is not recommended.
- Software is responsible for ensuring that code and data use do not result in the CAR being evicted.

2.4.5 Validated CAR size

In APL, the L2 cache size might be limited to 512kB on some SKUs thus it is recommended to have the CAR size limited to 512kB. But it is OK to check the real L2 size and use up the whole L2 for CAR.

One thing to keep in mind is that the CAR would be used for both code and stack. It is recommended to leave 128kB for stack (thus 384kB available for code in 512kB CAR size configuration). This holds true for the bigger CAR size as well.

2.5 MTRR Programming Considerations

In addition to setting the system memory address map, the IA FW should also program the MTRR (Memory Type Range Registers), which are part of the processors. The following is the suggested MTRR settings for various memory regions. The IA FW has control of 6 MTRR registers and the OS has control of 2 MTRR registers. Below table shows an example of the following configuration:

Table 2-1. APL Cache Modes

Core 0, Thread 0 CR0.CD	Core 0, Thread 1, CR0.CD	Core 0, IA32_MTRR _DEFTYPE.E	Core 0 usage of L1I,L1D MLC and LLC	Core n, IA32_MTRR _DEFTYPE.E	Shared LLC Cache State (excluding core 0)
1	1	1	Disabled	1	Available
				0	Disabled
		0		1	Available
				0	Disabled (all)
	0	1	Unsupported	1	Available
				0	Disabled
		0		1	Available
				0	Disabled
0	1	1		1	Available
				0	Disabled
		0		1	Available
				0	Disabled
	0	1	Enabled	1	Available
				0	Disabled
		0	Disabled	1	Available
				0	Disabled (all)

Table 2-2. Memory Map and MTRR Programming Example

Memory Range	Description	MTRR Setting	MTRR Encoding	MTRR Used for Configuration
0h-9FFFFh	640K Main memory	WB	06	MTRRfix64K_0000 MTRRfix16K_8000
A0000h – AFFFFh	64K Compatible SMRAM Segment	WP	05	MTRRfix16K_A0000
B0000h – BFFFFh	64K Compatible SMRAM Segment	WP	05	MTRRfix16K_B0000
C0000h – C7FFFh	Expansion ROM	WP	05	MTRRfix4K_C0000
C8000h – CFFFFh	Expansion ROM	WP	05	MTRRfix4K_C8000
D0000h – DFFFFh	Expansion ROM	WP	05	MTRRfix4K_D0000 MTRRfix4K_D8000
E0000h – EFFFFh	Extended System IA FW	WP	05	MTRRfix4K_E0000 MTRRfix4K_E8000
F0000h – FFFFFh	64K System IA FW	WP	05	MTRRfix4K_F0000 MTRRfix4K_F8000
1M – TOLM	Extended Memory	WB	06	Variable MTRR
111M – 112M	Extended Memory used as VGA memory space and TSEG memory space	UC	00	Variable MTRR **NOTE:** The TSEG region will also be declared as UC in this case. Refer processor documentation on how to account for speculative prefetch executions.
112M-128M	Extended Memory used as VGA memory space	UC	00	Variable MTRR (Need to use another MTRR due to power of 2 restrictions for programming the MTRR's)
FEDA0000h – FEDBFFFFh	HSEG memory region	UC	00	Variable MTRR

§

3 Multiprocessor Initialization

In a multiprocessor (MP) environment, each processor is responsible for the proper initialization of all items described in Chapter 1.7. It is possible that some configuration parameters may be different between each of the processors in an MP system (e.g. the CPUID may indicate a different stepping, and require a different microcode update). This section describes algorithms to direct the development of the multiprocessor of APL SoC.

Note: Intel multi-core processors must also follow the multiprocessor Initialization guidelines described in this chapter.

3.1 Multiprocessor Tables in IA FW

It is important to note that in order to support older multiprocessor operating systems that are not fully compliant with the ACPI spec, the IA FW is required to build the multiprocessor table, and must continue to be compatible with the Multiprocessor Specification, Version 1.4. Newer operating systems require that the IA FW comply with the ACPI Specification by providing multiprocessor information. MP table entries must be built for all active processors and ACPI table entries must be built for all active processors (including ALL logical processors). For Performance reasons, the ACPI table must be constructed so that all primary logical processors are listed before any of the secondary logical processors (or primary cores before secondary cores).

Package 0, Core 0, Logical Processor 0 BSP

Package 0, Core 1, Logical Processor 0 AP

Example 3-1. Processor Table Construction for MPS and ACPI

```
// The following pseudo code shows how to order the MPS and ACPI tables

FOR (T = 0; T< number of Threads per core, T++)
{
  FOR (C = 0; C < number of cores per package, C++)
       {
    FOR (P = 0, P < number of processor packages, P++)
        {
          // Place the processor in the ACPI
  // Multiple APIC Description Table (MADT)

          Add the processor identified by P,C,L to the ACPI MADT

           IF (T = 0)
          {
                  // Only the first logical processor per core
                  // is added to the MPS Table
```

```
                Add the processor identified by P,C,L to the MPS Table
        }
         }
       }
}
```

Some operation systems will not recognize or use any Application Processor whose entry in the MP table or the ACPI table appears prior to the Boot-Strap Processor entry. Therefore it is required that the BSP always appear first in the MP table, and in the ACPI table.

Note: Although the APIC ID on MP capable processors can be changed by the IA FW, commercial OSes do not require sequentially numbered APICs. Therefore the IA FW is usually not required to renumber the local xAPIC except in large-scale MP platforms. The APIC ID register in UP and DP capable processors are read only and cannot be changed by software.

Note: All system IA FW must build an MP table. This includes uniprocessor systems using an 8259A equivalent PIC. Refer Section 3.6 of the *Multiprocessor Specification*, Version 1.4 for a detailed description of the MP table contents and PIC mode options.

3.2 Startup Inter-Processor Interrupt (SIPI)

On the processor the IA FW must broadcast an INIT-SIPI- SIPI to signal the APs to begin the MP initialization. It is required that the vector for each SIPI be identical for the code to remain portable between processor families.

After RESET# and after receiving an INIT IPI, the APs enter the Wait for SIPI state. In the Wait for SIPI state an AP will not respond to an SMI#, NMI or INTR assertions. However, NMI and SMI# assertions will be pended. When an AP receives a SIPI, any pending NMI or SMI# will be serviced before handling the SIPI. To prevent pended NMI or SMI for application processors, each AP when it completes its MP initialization sequence must be left in an HLT state with interrupts disabled. In the HLT state the processor will respond to STPCLK#, SMI#, and NMI assertions. Thus, the IA FW must maintain a valid stack, real-mode interrupt vector table, and valid NMI handler to allow each AP to handle an NMI assertion in a determinate fashion. NMI handlers for all of the APs can simply be an IRET instruction, assuming the BSP completely processes the NMI. If the IA FW implements a common NMI handler used by the BSP and all of the APs, then the NMI handler must invoke a locking mechanism to prevent simultaneous processing in the NMI handler. The memory used for the AP NMI handler and each AP's unique stack must be reserved for the IA FW. Also the IA FW must maintain a valid and unique SMI entry point for each processor.

3.3 Bootstrap Processor Selection

In Intel multiprocessor systems, the MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs. The BSP selection mechanism will set the BSP flag in the

X86_CR_APICBASEMSR of the BSP, indicating that it is the BSP. This flag is cleared for all other processors.

In APL SoC, the BSP will be determined by a combination of the RESOLVED_CORE_VECTOR value and the WHO_AM_I value. Within a CMP module, the enabled core with the lowest ID (from WHO_AM_I) will be the Package BSP. This core will be responsible for initializing shared state within the CMP. The other core (if enabled) is an Application Processor and will enter Wait-For-SIPI after performing its own initialization.

Note: Throughout this document, the use of the term BSP refers to the initial active logical processor of the elected bootstrap processor and the term application processor refers to the alternate logical processor on the bootstrap processor as well as all other logical processors in the system.

3.4 Multiprocessor Initialization Algorithm

This section describes the algorithms that are used for initializing CMP enabled processors and multiprocessor systems. The first section is the BSP portion of the algorithm, the second section is the AP portion, and the third section describes selecting the lowest featured processor as the BSP.

Even a uniprocessor system that contains a multi-core processor must execute the MP initialization algorithm (including but not limited to SMM initialization, construction of the MP and ACPI tables, cache enabling).

3.4.1 Intel CMP Implications

Intel CMP features add a new dimension to the multiprocessor algorithm. For the most part, the additional logical processors appear to the IA FW simply as physical processors. The concept of sharing resources, however, does cause the IA FW to handle a number of things differently.

Most notably, the processor caches that are shared require all logical processors to have their cache enabled before any of the logical processors will actually begin caching. In the early stages of the IA FW boot process, it is recommended that cache be enabled in order to speed up the POST. Now with CMP enabled processors, an additional requirement is placed upon the IA FW that did not exist in the previous family of Intel processors. The IA FW must wake all application processors (AP's) in order for them to enable their MTRRs and caches. After this initialization is completed, the AP can be placed back into an MWAIT State to complete their multiprocessor initialization later in the boot process. In this state the AP's can be awakened with the INIT-SIPI-SIPI inter-processor interrupt sequence.

3.4.2 Multi-Core Implications

The multi-core processors add a new dimension to the multiprocessor algorithm. The additional processor cores appear to the IA FW simply as physical processors. Since most MSRs are not shared between cores, the IA FW must treat a multi-core processor as though it were multiple packages and initialize each logical processor on each core.

3.4.3 Algorithm for Bootstrap Processor Initialization

The Hardware-elected bootstrap processor starts executing the IA FW.

The bootstrap processor proceeds through POST (including initializing memory and cache).

The bootstrap processor loads its microcode update data to the processor.

The bootstrap processor initializes the MTRRs and enables cache. From this point forward RAM is available, the IA FW is running shadowed, and cache may be enabled.

Note: Other logical processors must be cache-enabled before the caching will operate.

Disable L1 and L2 processor caches. Refer Section 2.2. Initialize all MTRRs Maintain MTRRdeffType[11]= 0 to prevent L1 caching during this initialization.

Enable L1 and L2 processor caches. Refer the algorithm in Section 2.3

The bootstrap processor runs CPUID to determine its features for the lowest feature BSP determination later in the algorithm.

The bootstrap processor constructs its entry in a common database from its CPUID, MTRRs, Local APIC ID, and so on.

MP table entries and ACPI information are initialized for the bootstrap processor. The Intel Hyper-Threading Technology feature has an implication on the processor entries in these tables: the MP table must only include entries for the primary logical processors on each core; the ACPI table must include all logical processor entries. This will allow the Intel Hyper-Threading Technology enabled processors to continue to work properly with legacy operating systems as well as ACPI compliant operating systems.

The elected bootstrap processor enters the MP algorithm code.

The bootstrap processor initializes a pre-defined RAM location, the CPU counter, to a value of 1. The bootstrap processor initializes a “Lock Semaphore” to Vacant = 00h for use by a module that initializes all application processors.

The bootstrap processor waits for the application processors to complete their initialization (described in detail in the next section). This wait loop can be implemented in various ways. It can be use the interval timer (a hardware timer for use by the IA FW that is implemented by all Intel chipsets) to implement an exact delay time loop. The local xAPIC of the bootstrap processor also contains a timer that can be used for wait loop implementation. Or the timestamp counter can be used.

The bootstrap processor sends a STARTUP IPI message broadcast to all other processors (and logical processors) by programming the interrupt command register with a vector that points to the module that initializes the application processors.

The bootstrap processor waits for the application processors to complete their initialization (described in detail in the next section). This wait loop can be implemented in various ways. It can use the interval timer (a hardware timer for use by the IA FW that is implemented by all Intel chipsets) to implement an exact delay time loop. The local xAPIC of the bootstrap processor also contains a timer that can be used for wait loop implementation. Or the timestamp counter can be used.

3.4.4 Algorithm for Application Processor Initialization

All application processors wake up simultaneously after receiving the STARTUP IPI broadcast xAPIC message from the bootstrap processor and start executing this module.

The first processor that executes the "Test Lock" procedure sets the "Lock Semaphore" routine to NotVacant=0FFh and proceeds to Step 3. All other processors wait in the "Test Lock" procedure until the processor that moved to Step 3 completes and unlocks the lock semaphore variable. The "Test Lock" procedure can be coded as below pseudo code:

Example 3-2. Race to Test Lock Semaphore

```
; A spin-wait loop using a ''test, test-and-set'' technique to determine
; the availability of the synchronization variable.
; This technique is recommended when writing spin-wait loops to
; run on IA-32 architecture processors.

Test_lock:
    CMP     Byte Ptr [LockSemaphore], 0             ; Is the lock free?
    JE      Get_lock
    PAUSE
    JMP     Test_lock

Get_lock:
    MOV     AL, NotVacant
    XCHG    Byte Ptr [LockSemaphore], AL            ; Try to get lock.
    CMP     AL, 0                                   ; Test if successful

    JNE     Test_lock
Blocking:
Continue to Step 3
```

The first application processor to emerge from the "Test Lock" procedure increments the CPU counter variable.

The application processor loads its microcode update data to the processor.

The application processor initializes the MTRRs from a common database (remember that all processor MTRRs must be identical, and in order to ensure that all processors and logical processors have their MTRRs programmed identically, they should be initialized directly from the database).

Disable L1 and L2 processor caches. Refer Section 2.2 (Disabling the Caches)

Initialize all MTRRs. Maintain MTRRdefType[11]=0 to prevent L1 caching during this initialization.

Enable L1 and L2 processor caches. Refer the algorithm in section 2.3 (Re-Enabling the Caches).

The application processor enables its cache (Remember: the Intel Hyper-Threading Technology architecture prevents caching from operating until all logical processor's CR0.CD bits are cleared).

The application processor runs CPUID to determine its features.

MP table entries and ACPI information should be initialized for the application processor now. The Intel Hyper-Threading Technology feature has an implication on the processor entries in these tables: the MP table must only include entries for the primary logical processors; the ACPI table must include all logical processor entries. This will allow the Intel Hyper-Threading Technology enabled processors to continue to work properly with legacy operating systems as well as ACPI compliant operating systems.

The application processor enters SMM to relocate its SMBASE register and to initialize the SMI handler. Note that the SMBASE register is unique per logical processor. All logical processor SMBASE registers must be initialized.

Note: Prior to initializing the SMBASE register, the BSP must copy the SMBASE relocation code to 3000:8000, and the runtime SMI handler must be copied into its runtime location.

The application processor releases the lock on the semaphore by executing the “Release Lock” routine. This action allows the next waiting processor into the initialization loop:

Example 3-3. Releasing Lock Semaphore

```
ReleaseLock:
    MOV     AL, Vacant
    XCHG Byte Ptr [LockSemaphore], AL

Continue with step 11
```

The application processor disables interrupts, executes a HLT instruction and waits for an INIT-SIPI-SIPI message from the operating system.

Note: The application processor must not be placed in the “Wait for SIPI” state in this step.

3.4.5 Lowest-Feature Processor as Bootstrap Processor

The “Lowest-Feature” processor is defined as the processor whose features, as reported by the CPUID instruction's feature flags and extended feature flags, are common to all other processors in the system. In the event of a tie in features, or no processor matches the common set of feature flag, the processor with the lowest actual CPUID is the lowest featured processor. Intel requires that the lowest-feature processor in an MP system be selected as the bootstrap processor.

Once the bootstrap processor has determined that all of the application processors have completed initialization, the following steps must be taken to enable the lowest-feature processor as the bootstrap processor:

The feature flags for all operational logical processors are checked -- the feature flags are stored for each operational processor under its entry in the common database table. During application processor initialization, each non-bootstrap processor loads the feature flags information in the table using the CPUID instruction. The bootstrap processor feature flag information is constructed as part of early POST, again using the CPUID instruction.

Check if the current bootstrap processor is the lowest-feature processor, if so go to step 7. The lowest-feature can be determined as follows: Using the database table, the IA FW should AND together the 64-bit Feature Flag information (returned in the ECX:EDX register pair when the CPUID instruction is executed with EAX = 1) and the additional 128-bit Extended Feature Flag information (returned in the EAX:EBX:ECX:EDX registers when the CPUID instruction is executed with EAX =80000001h) for all the operational processors. The resulting mask is compared to the 192-bit Feature Flag information for each individual processor. If there is a match, the corresponding processor is the lowest-feature processor. There may be cases where more than one processor is the lowest-feature processor. For these cases IA FW checks the processor signature information and enables the oldest processor (the one with lowest family, model and stepping) as the BSP.

If the current bootstrap processor is not the lowest-feature processor, the bootstrap processor clear its BSP bit in the X86_CR_APICBASE register.

The bootstrap processor initiates a STARTUP IPI (the destination being the new lowest-feature BSP) and programs the Interrupt Command Register with a vector for the new bootstrap processor to execute.

The original bootstrap processor disables interrupts and executes an HLT instruction.

After receiving the START IPI, the new bootstrap processor must set its BSP bit in the X86_CR_APICBASE register.

The bootstrap processor proceeds through the rest of the IA FW initialization.

Note: To determine the BSP (Boot-Strap Processor) utilizing the LFP (Lowest-Feature Processor) algorithm, if the resulting feature flags after ANDing the 192-bit feature flags from the installed processors do not result in matching with the features flags of at least one of the processors, IA FW needs to set the processor with the lowest processor signature (i.e., Family Model and Stepping) as the bootstrap processor.

Features which are not common to all processors should not be enabled on those that support uncommon features.

3.4.6 Deterministic Bootstrap Processor

In some systems, it may be necessary for the IA FW to enable a specific processor as the BSP. Before enabling a specific processor as the BSP, the IA FW must execute the lowest feature processor algorithm. After completing the lowest feature processor algorithm, the IA FW may enable a specific processor as the BSP only if that processor exactly matches the feature set and CPUID of the processor chosen by the lowest feature processor algorithm. If the feature set and CPUID do match, the IA FW must not enable a new BSP. Failure to follow these guidelines may result in the OS attempting to use instructions or other features not implemented on the processor chosen by the lowest feature algorithm. Execute the following steps to set a specific processor as the BSP. These steps correspond to Steps 3 through 7 of the lowest feature processor algorithm.

If the desired processor matches the feature set and CPUID of the current bootstrap processor, the current bootstrap processor must clear its BSP bit in X86_CR_APICBASE register.

The current bootstrap processor initiates a STARTUP IPI (the destination being the desired processor's APIC ID) and programs the Interrupt Command Register with a vector for the desired BSP to execute.

If the desired processor does not respond to the SIPI within 100 microseconds, the current BSP must re-set its BSP bit in the X86_CR_APICBASE register and exit this algorithm.

The current bootstrap processor disables interrupts and executes a HLT instruction.

After receiving the START IPI, the desired BSP set its BSP bit in X86_CR_APICBASE register.

The new bootstrap processor must re-order the MP table and ACPI table and then proceed through the remainder of the IA FW initialization.

3.4.7 Determining Primary Logical Processor

The general case for detecting the primary logical processor within a processor package can be based on the Default APIC ID and the Maximum Logical Processor Count (which can also be interpreted as the number of APIC IDs reserved for this package) and works for any number of logical processors.

Both the Default APIC ID and the maximum number of logical processors per package are available from the execution of the CPUID instruction with the value of 1 in the EAX register. After execution of CPUID instruction with EAX = 1, the maximum number of logical processors per package is in EBX bits [23:16] and the Default APIC ID is EBX [31:24].

The maximum number of logical processors per package can also be considered the number of APIC ID values reserved for this package. The number of APIC IDs reserved for the package must be a power of 2 to allow for unique socket ID values. So a mask for the intra-package APIC ID can be created by subtracting 1 from the maximum number of logical processors per package value.

Example 3-4. Physical Processor - Primary Logical Processor Detection

```
General Case

Execute CPUID function 1
Default APIC_ID = EBX[31:24]
Maximum logical processors per package= EBX[23:16]
Intra Package Mask = Maximum logical processors per package - 1
IF (Default APIC ID & Intra Package Mask) == 0)
        Primary Logical Processor
ELSE
        Alternate Logical Processor
```

3.4.8 Determining Primary Logical Processor (Physical Processor)

```
; Finding Physical Package ID and Detecting Primary logical processor.
; This algorithm is not limited by the number of logical processors.
Mov eax, 1
CPUID

; After CPUID (EAX = 1) the following are valid:
; EBX [31:24] = Default APIC ID
; EBX [23:16] = Logical Processor Count
; EDX [28]  = Hyper Threading Technology Flag

BTEDX, 28t       ; Check if Hyper Threading Technology supported
JNC    no_HT             ; Jump if not

BSWAP  EBX               ; bl = Default APIC ID
                         ; bh = Logical Processor Count

MOVZX  EAX, BH           ;eax = Logical Processor Count
CMP    EAX, 1            ; Is the Logical Processor count one?
JZ     no_HT             ; Jump if Hyper-Threading is disabled
; At this point Hyper-Threading Technology is supported.
; Algorithm:
; Logical Processor Mask width = 1+MSB(Logical Processor Count-1)
; Logical Processor Mask = Mask with (width) LS Bits set to 1
; Package ID = Default APIC ID AND (NOT Logical Processor Mask)
;
; All logical processor with Default APIC ID equal to Package IS is the
; Primary Logical Processor
;
; Employ the Logical Processor Mask width algorithm
; Logical Processor Mask width = 1+MSB(Logical Processor Count-1)

DEC    EAX               ; eax = Logical Processor Count-1
BSR ECX, EAX             ; bsr returns MS bit position
INC    ECX               ; ecx = Logical Processor Mask Width

; Use Logical Processor Mask Width to obtain the
; Primary Logical Processor's APIC ID (Package ID)

MOVEX EAX, BL            ; eax = Default APIC ID
Shr    eax, cl
SHL  EAX, cL             ; eax = Default APIC ID AND Logical Processor Mask
```

```
; EAX = Primary Logical APIC ID or Package ID
; All processors with the same Package ID (EAX)are
; In the same processor core.

CMP  AL, BL ; Is Package ID (AL) = default APIC ID (BL)?
        ; Zero flag indicates Primary (ZP) or Secondary(NZ)
```

§

4 xAPIC

The APL SoC contains an on-chip local Advanced Programmable Interrupt Controller (APIC) compatible with the APIC implementation of Intel's Core™ Micro-architecture based processors. Similar to the Local APIC on previous processors, the APL SoC Local APIC is referred to as the Local xAPIC.

The Extended APIC ID in CPUID function 0Bh, CPUID. (EAX=0Bh):EDX[31:0], will report the full 32 bit ID for each logical processor. This allows IA FW to determine if a system has processors with IDs exceeding the 8-bit initial APIC ID limit the CPUID function 1, CPUID.(EAX=01h):EBX[31:24], reports. If the values of CPUID.(EAX=0Bh):EDX[31:0] reported by all logical processors in a system are less than 255, IA FW is recommended to transfer control to OS in legacy xAPIC mode with no visible option in setup menu. Refer the IA-32 Intel® Architecture Software Developer's Manual, Volume 3A: System Programming Guide, for a complete description of the Local xAPIC

4.1 X86_CR_APICBASE

The APIC Base Register (X86_CR_APICBASE) contains the APIC base address, permitting the relocation of the APIC memory map. The APIC Base Register also contains the APIC Global enable bit, APIC Extended Mode enable bit and Boot Strap Processor (BSP) indicator bit.

Refer EDS Spec for the register descriptions.

4.2 Legacy xAPIC Mode

Legacy xAPIC Mode is fully described in the IA-32 Intel® Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Refer this document for a complete description of the Local xAPIC.

4.2.1 Legacy xAPIC Mode IA FW Requirements

The IA FW should pass control to the OS in legacy mode when the APIC IDs of all logical processors in the system are less than 255. The setup option for switching to extended xAPIC mode should be hidden and invisible to users.

To operate in the legacy xAPIC interrupt model, the following steps should be added to the IA FW:

- Local xAPIC must be software-enabled and assigned a memory range that is both in an uncacheable memory region and does not conflict with motherboard.
- APIC Base Address (Bits[39:12] in the X86_CR_APICBASE Register) should not be remapped.
- If the IA FW is to pass control to the OS in legacy mode on a system that has APIC IDs greater than 255 because the user has indicated via a IA FW setup option that legacy mode should be forced, then the IA FW should:

- Attempt to relocate all APIC IDs below 255. IA FW must relocate all IDs within a package as a block of IDs. It is recommended that the block of IDs which includes 255 not be used. When building the Multiple APIC Descriptor Table (MADT) in extended xAPIC mode, the IA FW must verify that both the 32-bit and 8-bit addresses are unique. If the IA FW is unable to relocate all the APIC IDs below 255, the IA FW must disable the xAPIC on all processors with ID of 255 or greater before placing the processor in a MONITOR/MWAIT loop or other power saving state to ensure the processors do not respond to a INIT SIPI SIPI sequence intended for another processor.
- Use the legacy processor entries in the ACPI table that support only processor IDs up to 255.

Note: Since reads to the Local APIC registers do not generate external bus cycles, the IA FW should map the Local xAPIC for all logical processors in the system to the same address. It is recommended that the power-on default address of 0FEE00000h be maintained whenever possible.

- Local APIC must be initialized in virtual-wire mode.
- I/O APIC must be enabled via PCI configuration space accesses.
- Plug and Play and ACPI configuration interfaces must properly indicate to the OS that the memory range used by both the local and I/O APICs, are consumed by the platform.
- ACPI code must add a Multiple APIC Description Table.
- ACPI code must add interrupt routing to account for I/O APIC routing difference.

4.3 Extended xAPIC Mode (x2APIC)

The processor is Extended xAPIC Mode capable only when the "Extended xAPIC Mode" feature flag is set, CPUID. (EAX=1):ECX[21] == 1. The extended xAPIC Mode uses an MSR interface with MSRs 800h to 8FFh reserved for the xAPIC registers. The MSR interface is enabled by setting the "APIC Extended Mode" enable bit (Bit 10) in the X86_CR_APICBASE register. The MSR interface maintains a 1:1 register MMIO offset and MSR address offset except for elimination of the Destination Format register and the Interrupt Command Register (ICR) which is now combined into a single MSR. This combining of the ICR results in a single MSR write to dispatch inter-processor interrupts (IPIs).

The upper 32 bits of all MSRs (except ICR) are reserved. The MSR interface extends the APIC IDs from 8 bits to 32 bits, and extends Logical APIC IDs to use 16 bits for the cluster and 16 bits for the intra-cluster logical ID.

Modification of any reserved bit in the xAPIC MSRs will result in a #GP fault being generated. A #GP will also be raised if an xAPIC MSR access is attempted when either CPUID. (EAX=1):ECX[21] == 0 and when the "xAPIC Extended Mode" enable bit is cleared (X86_CR_APICBASE [10] == 0).

The WRMSR instruction serializing semantic is relaxed for writes to the extended xAPIC MSRs. Software must not use WRMSR instructions which access extended xAPIC MSRs as a serializing instruction.

Access to the xAPIC MSR space via WRMSR or RDMSR instructions while in legacy mode will result in a #GP fault being generated.

4.3.1 Requirements for Extended xAPIC Mode

On an extended xAPIC capable system, IA FW must provide a setup option to disable the extended xAPIC mode to support legacy operating systems.

Extended xAPIC Mode “Disabled” (default setting): IA FW must ensure that only logical processors with CPUID. (EAX=0Bh).EDX value less than 255 are enabled. IA FW initialization on all logical processors with CPUID.(EAX=0Bh).EDX values greater than or equal to 255 must (a) disable APIC and execute CLI in each logical processor, and (b) leave these logical processor in the lowest power state so that these processors do not respond to INIT IPI during OS boot. The BSP and all the enabled logical processor operate in xAPIC mode after IA FW passed control to OS. Application requiring processor topology information can use OS provided legacy services based on 8-bit initial APIC IDs or legacy topology information from CPUID.(EAX=01h) and CPUID.(EAX=04h) leaves.

Extended xAPIC Mode “Enabled”: IA FW enables x2APIC in Boot Strap Processor (BSP) and all Application Processors (AP) before passing control to the OS. Application requiring processor topology information must use OS provided services based on x2APIC IDs or CPUID.0BH leaf.

The IA FW should only pass control to the OS with xAPICs in extended mode if there are processors with APIC ID greater than or equal to 255, Interrupt Remapping is enabled and the user has enabled extended xAPIC mode in IA FW setup. In all other cases, IA FW should pass control to the OS with xAPIC in legacy mode.

To operate in the extended xAPIC interrupt model, the following steps should be added to the system IA FW:

- IA FW should use the ACPI table entries (MADT, SRAT, DMAR) that support full 32-bit processor IDs.
- Declare processors with APIC IDs < 255 as Processor() objects as defined in the ACPI specification.
- Declare processors with APIC IDs >= 255 as Device() objects as defined in the ACPI specification.

4.3.2 Restrictions for Extended xAPIC Mode

The extended xAPIC MSRs follow the RZ semantics for reserved bit - non-zero writes will result in a GP fault and reads will return zeros. RDMSR and WRMSR operations to reserved MSRs will raise a GP fault.

Refer Intel® 64 Architecture x2APIC Specification for a complete description of the Extended xAPIC.

4.3.3 Extended xAPIC Registers

Modification of any reserved bit in the xAPIC MSRs will result in a #GP fault being generated. A #GP will also be raised if an xAPIC MSR access is attempted when either CPUID(EAX=1):ECX[21] == 0 or when the “APIC Extended Mode” enable bit is cleared (X86_CR_APICBASE [10] == 0).

§

5 System Management Mode (SMM)

System Management Mode (SMM) is a high priority mode of the processor that is entered when the processor acknowledges a System Management Interrupt (SMI). When the processor enters SMM it writes the current state of the processor (the processor's context) to memory and begins executing specialized IA FW code. When entering SMM the processor in a mode similar to real-address mode except there are no privilege levels or address mapping.

The APL SoC introduces new capabilities and requirements for SMM. This chapter describes the IA FW changes required to support SMM in the APL SoC.

For a detailed overview of SMM refer Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.

5.1 Initializing SMM

During the POST, the bootstrap processor (BSP) is responsible for:

- Configuring the chipset controls for the SMM region. This must be the Top of Low Memory Segment (TSEG) for the APL SoC.
- Copy the runtime SMM handler to the TSEG.
- Copying the SMM relocation handler to 30000h + 8000h.
- Invoking a SMI IPI to self to execute the SMM relocation handler.
- Resume execution of the POST.

Since the BSP initialized the chipset and installed the runtime SMM handler, the application processors (APs) must:

- Send an SMI IPI to self to execute the SMM relocation handler. This first SMI will cause the processor to vector to 30000h + 8000h.
- Resume execution of the multiprocessor initialization.

5.1.1 Responsibilities of SMM Relocation Handler

For each logical processor, the first SMI it receives after a reset will cause the processor to vector to 30000h + 8000h. This is where the IA FW installs the SMM Relocation Handler. The SMM relocation handler is responsible for:

- Assigning each processor a unique SMBASE. The SMBASE is only configurable while in SMM and must be written to the SMBASE register which is part of the processor context saved when the SMI was acknowledged. The SMBASE register during this first SMI is located at memory address 30000h + (8000h + 7EF8h). Subsequent SMIs will cause the processor to vector to the new SMM entry point which is SMBASE + 8000h. The minimum distance between adjacent SMBASE assignments is 400h (1024) bytes.

- Configuring the System Management Mode Range Registers (SMRR). The SMRR maps the cacheability of the runtime SMM handler. The runtime handler must be located in TSEG. The SMRRs enhance SMM by making the SMM memory cacheable upon acknowledging an SMI. The SMRR is used to define the cacheability type of all of the TSEG. SMRAM base and size must be aligned on a power of 2 boundary. Since TSEG is now cacheable when the processor is writing its context, entry and exit of the runtime SMM handler should be faster.

During the POST, the IA FW must initialize the SMBASE for each logical processor in the physical processors. Each logical processor must be assigned a unique SMBASE. The SMBASE for a logical processor must maintain at least 400h address separation from another logical processor's SMBASE.

5.2 Enhanced SMM Features

The below Enhanced SMM features are available on Intel microarchitecture code name APL. For the above features, IA FW must test the associated capability bit before attempting to access any of the above registers. The capability bits are discussed in the register description.

- Determine if a thread is going to be delayed entering SMM using MSR SMM_DELAYED (4E2h) [11:0].
- Determine if a thread is blocked from entering SMM from MSR SMM_BLOCKED

  (4E3h) [11:0].
- Enable SMM code execution range checking by setting MSR FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL (4E0h) [2].
- Select whether SMM Save State is to be written to legacy SMRAM or to MSRs by

Configuring FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL (4E0h) [1].

- SMM Code Execution Range Checking

  Intel microarchitecture code name APL can be configured to do SMM code execution range checking and signal an machine-check if execution outside the range of the SMRR or SMRR2 is detected while SMI is active.
  - If the MSR UCODE_CR_SMM_MCA_CAP (17Dh) [58] is 1b, then SMM code execution range checking can be enabled by setting the MSR FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL (4E0h) [2] to 1b. SMM code execution range checking can only be enabled while in SMM, but range checking is not applied to the current SMI. It will be applied to subsequent SMIs. If code range checking is desired, then the SMM relocation handler should be responsible for enabling SMM code execution range checking.
  - If code range checking is to be enabled, then only the last thread to execute SMM relocation can set MSR FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL [2] to 1b. That is because the SMRRs will already point to the runtime SMM handlers and the SMM relocation handler executes outside those ranges.

- SMI Long Flow indicator

  If the MSR UCODE_CR_SMM_MCA_CAP (17Dh) [59] is set to 1, then Intel microarchitecture code name APL supports the SMM_DELAYED register. If a thread is in a long flow (e.g., WBINVD) when an SMI is signaled, then the corresponding bit in the SMM_DELAYED register will be set indicating to the SMM Monarch that a specific thread or threads will respond to the SMI but must finish the current long flow operation. The SMM handler can use the SMM_DELAYED register to assist in determining why a thread has not checked in to SMM.

SMI Blocked If a thread is in a state that prevents it acknowledging an SMI, Intel microarchitecture code name APL will indicate the thread is blocked by setting the corresponding bit in the SMM_BLOCKED register. The SMM handler can use the SMM_BLOCKED register to assist in determining why a thread has not checked in to SMM. When a thread is in a state where SMI is blocked, the SMI gets pended and will be serviced when the thread is no longer in a state where SMIs are blocked (e.g., the Wait-for-SIPI state).

5.3 SMM State Save Map

Upon entering SMM, the IA FW must check the SMMREV_ID register located at SMBASE + 8000h + 7EFCh. When MSR FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL (4E0h) [1] is 0b, the SMMREV_ID is located in the MSR SMM_REVID (C21h). APL SoC will support a SMMREV_ID of 30101h.

5.4 SMM Rev ID Encoding

The new encoding scheme is different from the old scheme where there was a single revision number. The new encoding scheme uses two components a major number and a minor number.

Table 5-1. SMM Revision ID Encoding

31	30:18	17	16	15:8	7:0
STM Generated	Reserved	SMBase Relo	I/O Restart	Major	Minor

Generally, different Major versions are assumed not compatible and Minor version changes are considered backward compatible for architectural fields within the scope of a given major version. Bit 31 indicates if the State Save was generated by STM.

5.4.1 SMM State Save Map in the Processor

Table 5-2. SMM State Save Map (SMM_REV_ID=3_0101h)

Processor Register Name	FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL [1] == 0 (State saved to Legacy SMRAM)	
	SMBASE + 800h + offset	Access
CR0	7FF8	SMM RO

Processor Register Name	FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL [1] == 0 (State saved to Legacy SMRAM)	
	SMBASE + 800h + offset	Access
CR3	7FF0	SMM RO
RFLAGS	7FE8	SMM R/SMM W
EFER	7FE0	SMM R/SMM W
RIP	7FD8	SMM R/SMM W
DR6	7FD0	SMM RO
DR7	7FC8	SMM RO
TR SEL	7FC4	SMM RO
LDTR SEL	7FC0	SMM RO
GS SEL	7FBC	SMM RO
FS SEL	7FB8	SMM RO
DS SEL	7FB4	SMM RO
SS SEL	7FB0	SMM RO
CS SEL	7FAC	SMM RO
ES SEL	7FA8	SMM RO
IO_MISC_INFO	7FA4	SMM RO
IO_MEM_ADDR	7F9C	SMM RO
RDI	7F94	SMM R/SMM W
RSI	7F8C	SMM R/SMM W
RBP	7F84	SMM R/SMM W
RSP	7F7C	SMM R/SMM W
RBX	7F74	SMM R/SMM W
RDX	7F6C	SMM R/SMM W

Processor Register Name	FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL [1] == 0 (State saved to Legacy SMRAM)	
	SMBASE + 800h + offset	Access
RCX	7F64	SMM R/SMM W
RAX	7F5C	SMM R/SMM W
R8	7F54	SMM R/SMM W
R9	7F4C	SMM R/SMM W
R10	7F44	SMM R/SMM W
R11	7F3C	SMM R/SMM W
R12	7F34	SMM R/SMM W
R13	7F2C	SMM R/SMM W
R14	7F24	SMM R/SMM W
R15	7F1C	SMM R/SMM W
Reserved	7F1B:7F04	
AutoHalt Restart	7F02	SMM R/SMM W
I/O Restart	7F00	SMM R/SMM W
SMREVID	7EFC	SMM RO
SMBASE	7EF8	SMM R/SMM W
Reserved	7EF7:7EE4	
EPTP_EN	7EE0	SMM RO
EPTP	7ED8	SMM RO
Reserved	7ED7:7EA0	
LDTR Base (low 32-bits)	7E9C	SMM RO
Reserved	7E98	

Processor Register Name	FUNNYIO_CR_SCP_SMM_FEATURE_CONTROL [1] == 0 (State saved to Legacy SMRAM)	
	SMBASE + 800h + offset	Access
IDTR Base (low 32-bits)	7E94	SMM RO
Reserved	7E90	
GDTR Base (low 32-bits)	7E8C	SMM RO
Reserved	7E8B:7E44	
CR4	7E40	SMM RO
Reserved	7E3F:7DF0	
I/O RIP	7DE8	SMM R/SMM W
Reserved	7DE7:7DDC	
IDTR Base (upper 32-bits)	7DD8	SMM RO
LDTR Base (upper 32-bits)	7DD4	SMM RO
GDTR Base (upper 32-bits)	7DD0	SMM RO
Reserved	7DCF:7C00	

Table 5-3. Save State Map Field IO_MEM_ADDR Definition

Bit	Description
63:0	**Starting Linear Address of Memory Operand:** For string I/O operations (including INS, REP INS, OUTS, REP OUTS), this 64-bit address is the initial memory operand address. For I/O instructions that do not reference memory, the value is zero.

Table 5-4. Save State Map Field IO_MISC_INFO Definition

Bit	Description
31:16	**I/O Port Address:** This value represents the port address encoded as an immediate or register value.

Bit	Description
15:8	Reserved
7:4	I/O Type: Bit Description of Instruction Encoding **7 Port Address:** A value of 1 indicates the Port Address was encoded within the instruction data. A value of 0 indicates the port address is contained in DX register. **6 REP Prefix:** A value of 1 indicates the instruction contains a REP prefix. A value of zero indicates the instruction contains no REP prefix. **5 String I/O Source or Destination:** A value of 1 indicates the instruction uses a string source or destination. A value of 0 indicates an accumulator register is used for source or destination. **4 Port Type:** A value of 1 indicates an IN. A value of 0 indicates an OUT.

Example Encoding for I/O Type Bit Field

Bits 7:4	Imm(1)/ Reg(0)	REP(1)/ Not(0)	String(1)/ Not(0)	IN(1)/ OUT(0)
IN imm	1	0	0	1
IN dx	0	0	0	1
OUT imm	1	0	0	0
OUT dx	0	0	0	0
INS	0	0	1	1
REP INS	0	1	1	1
OUTS	0	0	1	0
REP OUTS	0	1	1	0
MWAIT1	1	0	1	0

Bit	Description
3:1	**I/O Length:** This field provides the I/O instruction data length. Bits 3:1 I/O Length 001 Byte 010 Word 100 Dword
0	**Synchronous SMI:** Always 1 for a Synchronous SMI (sometimes 1 for Asynchronous SMI). A value of 0 indicates Asynchronous SMI. The SMI handler must examine the save I/O address to confirm the type of SMI. **NOTE:** The SMM handler is allowed to return to the same instruction which caused the SMI only when the SMI is found to be Synchronous. If the SMI is found to be Asynchronous, the handler must always return to the next instruction.

NOTE: Refer table below for the new register decoding for MWAIT type.

Table 5-5. Save State Map Field IO_MISC_INFO Definition—When I/O Type Is MWAIT

Bit	Description
31:28	**Package C-State (R/W):** The Coordinated C-State of the Package when the SMI was detected. Represents the target state if Synchronous SMI (bit 0) is 1 or the actual state if Synchronous SMI (bit 0) is 0. 1111b–C0 0000b–C1 0001b–C2 0011b–C4 0101b – C6 All others reserved on the APL SoC.
27:24	**Package C Sub State (R/W):** The sub state of the package C-State.
23:20	**Thread C-State (R/W):** The C-State of the thread when the SMI was detected. 1111b–C0 0000b–C1 0001b–C2 0011b–C4 0101b – C6
19:17	**L2 Shrink State (R/W):** L2 Shrink state of the thread when the SMI was detected. **001 –** Thread Full Shrink L2 **100 –** Thread No Shrink L2 All others reserved on the APL SoC.
16	**Thread Frequency (R/W):** The Frequency state of the thread when the SMI was detected. **0b –** Non – LFM Frequency **1b –** LFM Frequency All others reserved on the APL SoC.
15	**Block SMI (RO):** When set to “1b”, prevents SMI generation through North Complex upon C-State transitions.
14:8	Reserved
7:4	I/O Type: Bit Description of Instruction Encoding. Set to ‘1010b’ for C-State SMI.

Example Encoding for I/O Type Bit Field				
Bits 7:4	Imm(1)/Reg(0)	REP(1)/Not(0)	String(1)/Not(0)	IN(1)/OUT(0)
MWAIT	1	0	1	0

Bit	Description
3:1	Reserved

Bit	Description
0	**Synchronous CST SMI:** Set to 1 for a Synchronous CST SMI. When set to '1b', Monitor is preserved and SMI is synched to C-State resolution. When set to '0b', Monitor is preserved and MWAIT resulting in this SMI may or may not be restarted The SMM handler is allowed to return to the same instruction which caused the SMI only when the SMI is found to be Synchronous. If the SMI is found to be Asynchronous, the handler must always return to the next instruction.

5.5 SMRR Configuration Requirements

Configuring the SMRR registers only happens during the first SMI per logical processor where SMM relocation of the SMBASE is accomplished. SMRR configuration is NOT executed with each subsequent SMI.

Note: Configuration of the SMRR is discussed in Section 5.5.1. As discussed there, the SMRR can only be configured while the processor is in SMM and executing the SMM relocation handler.

While executing in SMM, the memory type of SMRAM is described by the SMRR. When executing outside SMM, if the memory request is made to an address mapped by the SMRR, it will always default to a UC memory access.

To minimize the number of variable MTRRs required to map conventional + low memory, the SMRR may overlap MTRRs. In the case where the SMRR overlaps MTRRs a decode precedence is applied such that the memory type is always defaulted to the operation of the SMRR.

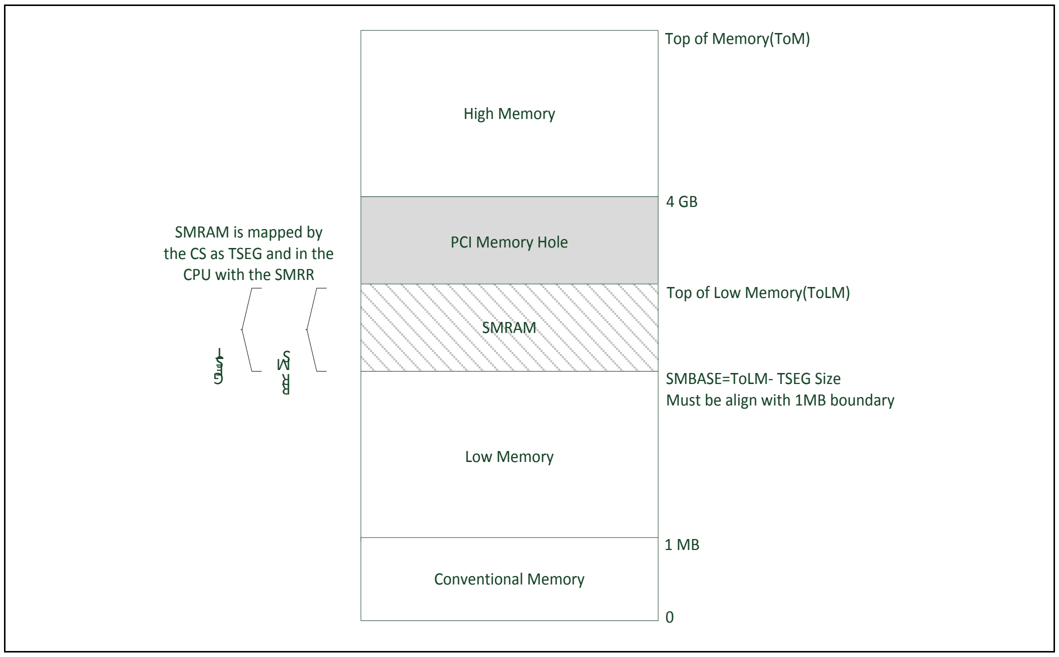
The cacheability type for SMRAM is defined by the SMRR. For best performance the SMRR must map SMRAM as WriteBack (WB) cacheable. IA FW must configure the IA32_CR_SMRR_PHYSBASE [2:0] = 06h for the WB cacheable type.

Note: The TSEG and the IA32_CR_SMRR_PHYSBASE must be aligned on an 8-MByte (or greater) power of 2 address boundary. The SMRR maps both the SMRAM region. Refer Figure 5-1.

Note: The TSEG size is not adjustable on APL platform, Intel support TSEG size as 8MB only in APL platform.

Note: The SMRR does not support the compatibility segment (CSEG).

Figure 5-1. SMRR Mapping with a Typical Memory Layout



5.5.1 IA FW Steps to Enable and Configure SMRR

Ensure all processor cores have microcode updates applied.

- Read the IA32_MTRRCAP register (MSR 0FEh) and test bit 11 of the IA32_MTRRCAP register. If bit 11 is set (1), then the SMRR is supported. If bit 11 is clear (0), the SMRR is not supported.
- Before setting the Lock Bit in the IA32_CR_FEATURE_CONTROL MSRregister (MSR 03Ah bit 0), IA FW must also set bit the SMRR_Enable bit (bit 3) in the IA32_CR_FEATURE_CONTROL MSR register on both cores. It is important that IA FW programs the SMRR_ENABLE the same way on all cores (all 1 or all 0), before accessing the SMRR_BASE/MASK registers or MTRRs, else behavior is undefined.
- If the IA FW has already configured Intel® VT then the IA FW must set the lock bit, otherwise IA FW must finish configuring Intel® VT before setting the lock bit. None of these bit settings take effect until the lock bit is set.

Note: Even though the SMRR is enabled by the above steps, the programming of the IA32_CR_SMRR_PHYSBASE register (MSR 1F2h) and IA32_CR_SMRR_PHYSMASK register (MSR 1F3h) must be completed while executing in SMM.

- Read IA32_MTRRCAP register (MSR 0FEh) and read the VCNT field to determine the available number of Variable MTRRs.

Note: IA FW must not assume a fixed number of variable MTRRs. On Atom ™, the total number of variable MTRRs is decremented by one when the SMRR is enabled. Failing to observe this and attempting to use a non-existent MTRR will result in a general protection fault.

- Using the available MTRRs describe the system memory map per chapter 3 of this document.

- The BSP copies the SMI handler to TSEG for the BSP and each AP
- Execute SMBASE relocation code (requires being in SMM), setting the new base equal to the TSEG + a unique entry offset.
- Configure the IA32_CR_SMRR_PHYSBASE using the same physical address as the beginning of the TSEG. Unlike a normal variable MTRR, the IA32_CR_SMRR_PHYSBASE register does not allow SW to specify a type.
- Configure the IA32_CR_SMRR_PHYSMASK with the same size as TSEG and set the valid bit (bit 11) prior to exiting the SMI handler.

Note: The IA32_CR_SMRR_PHYSBASE and IA32_CR_SMRR_PHYSMASK have the same alignment relationship as normal variable MTRRs, except that the upper 32 bits of the IA32_CR_SMRR_PHYSMASK register are zero.

Note: The SMRR feature should be disabled for BIOS SMM source level debugging.

5.5.2 Runtime Requirements and TSEG Caching

This section provides steps required to take advantage of the SMRR that was configured in the previous section.

The SMRR functions slightly differently than a normal variable MTRR. When enabled, the SMRR behaves as a UC type variable MTRR and prevents caching of TSEG. Unlike a normal variable MTRR, the SMRR cache-ability controls can only be written while executing in SMM. The SMI handler should dynamically disables the SMRR early in the handler, and re-enables it just before RSM.

To get the full benefit of the SMRR, the runtime SMI handler must satisfy the following requirements:

- The SMI entry point is contained within TSEG for all processors.
- The SMI handler must avoid execution of non-TSEG code. Furthermore, during SMI execution when the SMRR is disabled, the MTRRs must be configured such that the effective cache type of TSEG resolves to WT or UC. This means that when the SMRR is disabled, the following must be true:
- All enabled variable MTRRs that overlap the TSEG region must be WB, WT, or UC.
- WB TSEG must be avoided. In other words, if there is any WB MTRR overlaying any part of TSEG, it must be degraded by another MTRR that is either WT or UC for the portion of TSEG that would otherwise WB.
- If there is any portion of TSEG that is not covered by an enabled variable MTRR, the default MTRR must be either WT or UC.
- Any MTRRs that overlap TSEG and are not WB, WT, or UC must be disabled for the duration of time that the SMRR is disabled.

The following steps comprise one possible algorithm to satisfy the above rules without undue performance impact. This algorithm may be modified or optimized as long as the above rules are satisfied in all possible scenarios. Do not make any assumptions about the state of the variable MTRRs or the default MTRR on entry to SMM.

Generally, the algorithm below facilitates cacheable TSEG while executing in the runtime SMM handler, and un-cacheable when not executing in SMM. These steps must be taken on every processor.

1. Upon entry to SMM, make sure all processor cores are in SMM mode and only then do the followings for each processor core.
2. Select one variable MTRR and save the state of X86_CR_MTRRPHYSMASKn and X86_CR_MTRRPHYSBASEn.
3. Disable the selected MTRR by clearing X86_CR_MTRRPHYSMASKn.Valid (bit 11).
4. Copy the IA32_CR_SMRR_PHYSBASE and IA32_CR_SMRR_PHYSMASK into the selected variable MTRR.
5. Change the memory type of the selected variable MTRR to Write-Through.
6. Read the X86_CR_MTRRPHYSBASEn for all remaining MTRRs.
7. For every MTRR read in step 5 that overlaps TSEG and has a cache type other that WB, WT, or UC, do the following:
   – Read X86_CR_MTRRPHYSMASKn
   – Ensure X86_CR_MTRRPHYSMASKn.Valid (bit 11) is clear, disabling the MTRR.
8. Clear the IA32_CR_SMRR_PHYSMASK .Valid bit (bit 11) to 0.

Note: As long as there are no enabled UC MTRRs overlapping TSEG, the TSEG SMI handler will now executing cached using the Write-Through memory type. IA FW specific SMM code executes here. Non-TSEG addresses are cached according to the programming of the remainder of the MTRRs.

9. Complete execution of the SMI handler.
10. Set the IA32_CR_SMRR_PHYSMASK .Valid bit (bit 11) to 1.
11. Restore the state of the variable MTRR whose state was saved in step 1 and 7.
12. Wait for all processor cores are done with step 12.
13. Resume.

Note: Explicit flushing of the cache is not required.

5.6 SMM Handler Considerations

Configuring the SMRR registers only occurs during the first SMI per logical processor where SMM relocation of the SMBASE is accomplished. SMRR configuration is NOT executed with each subsequent SMI.

Various causes of SMI are routed through the SoC and signaled to each processor via an SMI message. Other SMIs are generated by a processor. That processor generated SMIs may be signaled only to the local processor (package or core).

IA FW SMM handler implementations vary. An implementation that expects a specific processor to service the SMI will need to be modified to allow an alternate logical processor (monarch) to service the SMI. This implementation change is required since the normal service processor may not be signaled if an SMI is local to a core or another processor package.

IA FW should determine if system share resources are required to service the SMI. If system share resources are required to service the SMI, then the IA FW elected monarch logical processor must signal an SMI to the other logical processors that have not yet entered SMM (checked in).

If system shared resources are not required to service the SMI, IA FW may elect to not signal the other logical processors that have not entered SMM (checked in).

IA FW should disable T-State Throttling (Refer MSR in EDS, "(1A0h)IA32_CR_MISC_ENABLES" bit [3]) while executing the SMM handler. If T-State Throttling is enabled and configured at lower duty cycles (Refer MSR in EDS "(19Ah) P_CR_THREAD_T_REQ" bits [3:0]), then it is possible for the SMI check-in monarch to exclude cores due to the check-in timeout expiring because the core has very little clock cycles to execute its code and check-in. This sync issue can cause further issues when an OS uses Machine Check Architecture, and can result in system shutdown.

Note: SMM handler should be aware that, as with other devices, DMA transactions to these devices, if in progress, will continue. Therefore, if such devices have to be used during SMM mode, care should be taken, such as not using devices for which a driver entity is active, by checking that the DMA controller is idle, and/or by waiting for the DMA controller to complete.

Example 5-1. SMM Handler Algorithm Outline, (Pseudo Code)

```
Assumptions:
1. Two main trees of SMI sources
  a. System Agent and PCH directed and signaled to all threads in all
packages
  b. CPU initiated and directed either to a specific thread or all threads
in a
     single package (Broadcast SMI IPI is the exception).

2. The first processor thread to enter SMM and checkin will become the
Monarch
  a. No thread specific execution handler, i.e., any thread can be the
monarch.
  b. The monarch can only access package shared MSRs (i.e., those MSRs that
are
          defined as shared between all logical processors in a single
package).
  c. The monarch can access CSRs from all packages.

3. If there is a valid CPU SMI cause detected but none from the CS, then
the other
        threads may need to be brought into SMM and the timer needs to be
reset.

4. The constant ''ExclusiveAccessCauseList'' is constructed by the IA FW
developer at
   the time the IA FW is compiled. It is assumed that the BIOS developer
knows if
   they need to access a shared resource in servicing the causes in this
list. Any
   SMI cause that does not need access to any share resource is left out
of this
   list.

Algorithm:
```

```
//
// At SMBASE + 8000 for each processor it vectors to the common handler.
// Before the far jump a thread specific stack may need to be configured.
//

IndividualSMIEntry::
  //
      // setup thread specific stack if needed
  //
      JMPF TSEG:CommonSMIHandler
-----------------------------------------------------------------
CommonSMIHandler::

CheckIn()                  // Grab checkin semaphore, elect monarch,
           // close exit gate, update checkin structure

IF ( monarch ) THEN
      GetCauses( CS, CPU ) // Read CS tree and CPU tree of SMI causes

      //
  // Determine how long to wait for other threads to check into SMM
      //
  TimeOut = CalculateTimeOut( ENTRY_RENDEZVOU )

  //
      // If no cause is set, this while loop is bypassed providing a means
  // for fast path through the SMI handler when no SMI cause is detected
      // (spurious).
  //
      // This handler does not wait for timeouts to expire when an unknown
  // SMI occurs.
//
While ( CPU_cause OR CS_cause ) DO
IPI_sent = FALSE
//
// Check if any of the current causes require exclusive access to
// system shared resources.  If a non-zero value is the result of
// the bitwise AND, then we must issue SMI IPI to bring the other
// threads into SMM so that IA FW can guarantee exclusive access
// (atomic access) to the shared system resource.  If exclusive
// access is not required, do not bring the other threads into SMM.
//
IF ((( CPU_cause OR CS_cause ) AND ExclusiveAccesCauseList) AND
  (IPI_sent == FALSE)) THEN
//
// An implicit assumption is made here.  If the SMI was
// delivered through the chipset, then all processor packages
// received the SMI.
//
IF (SoC IA core Processor) THEN
{
notIn = (checkin XOR registered)
restartTimer = (notIn != 0) // Flag that the timeout
// will need to be recalculated
// since SMI IPIs will be sent
WHILE ( notIn != 0 ) DO
BSF( id, notIn ) // Determine next ID
Send_SMI_IPI( id ) // Send directed SMI-IPI
BTR( notIn, id ) // Clear the bit in notIn
END WHILE
IPI_sent = TRUE // Only send IPIs once
//
// Since we just sent SMI IPIs see if we need to
// recalculate the wait timer
//

  IF ( restartTimer == TRUE ) THEN
       TimeOut = CalculateTimeOut( IPI_TIMEOUT )

       //
    // BIOS Implementation specific - IFF shared resource
       // are used then IA FW must wait for all processors to check
    // in or timeout before saving shared resources that will be
       // used by their SMI handler.  Examples CMOS (70h, 71),
    // Serial Bus controller …
       //
    WHILE ((checkin != registered) AND (CurrentTimer < TimeOut)) DO
       //
    // IA FW only waits if 1 or more threads have not
       // checked in. IA FW stops waiting if the TimeOut
    // occurs.  This might happen if a thread is in
       // Wait-for-SIPI
    //
                wait()
    END WHILE
                SaveSharedResources()// e.g., 70h, 71h …
    END IF

  END IF
END IF (exclusiveAccess)

WHILE ( CPU_cause != 0) DO
```

```
    ServiceCPUcause()                                                    //
The service routine also

       // clears the CPU status bit
END WHILE

WHILE ( CS_cause != 0 ) DO
      ServiceCScause()
      // The service routine also
                                                                 // clears
the CS status bit
END WHILE

//
// BIOS Implementation specific - IFF any shared resources are used
// then IA FW needs to restore resources used by the SMI cause
// handler. Examples CMOS (70h, 71) …
//

IF (IPI_sent)
  RestoreSharedResources()

//
// This wait is executed only if a CPU or CS cause was detected.
// EXIT rendezvous - If every processor has checked in then
// no wait penalty occurs, otherwise must wait for the TimeOut.
//

WHILE ((checkin != registered) AND (CurrentTimer < TimeOut)) DO
       //
    // IA FW only waits if 1 or more threads have not checked in.
       // IA FW stops waiting if the TimeOut occurs.  This happens
    // if a thread is in Wait-for-SIPI
       //
    wait()
      END WHILE
    //
       // Determine if another SMI came in while servicing the others.
    //
       GetCauses( CS, CPU )// Read CS tree and CPU tree of SMI causes
  END WHILE

  ClearCheckIn()
      OpenExitGate()

ELSE                                                                     //
non-monarch threads
```

```
  WHILE (ExitGate == CLOSED) DO
        wait()
  END WHILE

END IF

RSM
ExitSMI::
```

Example 5-2. Checkin()

```
Checkin() is a sequence of steps:
Obtain the checkin semaphore or wait until it is free
(use Test, Test-and-Set loop) to prevent starvation and promote thread
fairness
Determine if first thread to checkin
First thread to checkin becomes the Monarch (winner of race)
Monarch Closes the Exit gate
Monarch reads checkin timer (must be a monotonically increasing timer such
as the
PMTIMER)
Update the checkin structure(s)
Release the checkin semaphore
CHECKIN
; This sample UP/DP implementation assumes that the APIC ID is used as the
; indexed value into the checkin structure.  This example is limited to 32
; Logical processors.
;
; For example Gainestown the valid APIC IDs would be:
; Package 0: 00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
; Package 1: 10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h
    MOV    AL, 1 ; Get APIC ID from CPUID function 1
        CPUID
    BSWAP EBX ; Move the APIC ID into BL
        MOV EAX, DWORD PTR CS:[checkin]; Move checkin into a register so
    BTS EAX, BL ; modulo 32 is automatically applied
        MOV DWORD PTR CS:[checkin], EAX
```

Example 5-3. Semaphore Race Using Test, Test-and-Set

```
SEMAPHORE RACE

; A spin-wait loop using a "test, test-and-set" technique to determine
; the availability of the synchronization variable. This technique is
; recommended when writing spin-wait loops to run on IA-32 architecture
; processors.

Test_lock:
  CMP Byte Ptr [LockSemaphore], 0 ; Check if lock is free.
      JE Get_lock
  PAUSE
      JMP Test_lock
Get_lock:
      MOV AL, NotVacant
  XCHG Byte Ptr [LockSemaphore], AL ; Try to get lock.
      CMP AL, 0 ; Test if successful.
  JNE Test_lock
```

5.7 Recommended SMM Usage

5.7.1 Variables Support through SMM

This is the traditional approach followed by the BIG CORE platform. Before hand-off to OS the IA FW would lock the SPI access from host and also program the secure SPI access range from within the SMM value in the Flash descriptor. SPI controller on seeing the SMM would enable access to this variable window from SMM and enable protection once the SMM is returned.

UEFI Variables can be updated securely and in an isolated way by using SMM. SMM serves as the firewall for SPI Flash protection from Malwares running in non-SMM OS code. If SMM is not desired in the Platform, then a restricted Non-Volatile memory variable access can be achieved in the iLB CMOS. However, the CMOS area does not have the same level of protection from a Security standpoint, from Malwares, as the protection offered by SMM and standardized UEFI Variable access. CMOS accesses are not expected to be compliant with UEFI Variable definition and requirements.

§

6 IA FW SMIP

This chapter is intended for IA FW/BIOS engineer to understand IA FW **Signed Master Image Profile (SMIP)** definition and customization. IA FW SMIP contains platform-specific data that IA firmware/BIOS need to generate specific platform behavior. This allows customers to create the customized IFWI without modifying the IAFW/BIOS code. Refer "Apollo Lake SoC SPI and Signed Master Image Profile (SMIP) Programming Guide" for more detail about SMIP (Doc #559702). Below is the layout of IAFW SMIP.

Also, IA FW SMIP is an alternative way for customers to configure platform specific data based on different board design without re-compiling BIOS. However, customers still can disable IA FW SMIP in the IAFW/BIOS source code to ensure all the platform settings are there in IAFW/BIOS source code only.

SMIP Definition	SMIP Offset	Size in Bytes	Description
IafwSmipSignature	0	4	Value : 0xDEEDBEAF
IafwSmipVersion	4	1	IAFW SMIP Version for version control usage – Read Only
Reserved	5	0x1C1	Reserved
DRAM Configuration	0x1C6	0x4C2	This describes the DRAM Memory Configuration (Rank, Channel) This is equivalent to SPD configuration Which includes Memory geometry configuration
Reserved	0x688	2	Reserved
Platform Board Configuration	0x68A	0xBC4	GPIO Configuration
IafwSmipFooter	0x124E	4	Value: 0xAAFFCCAA
Checksum	0x1252	1	

6.1 DRAM Configuration

DRAM Configuration is enabled on Apollo Lake. However, as we mentioned earlier, IAFW/BIOS programmer still can skip DRAM configuration of IAFW SMIP and hardcode all the memory related parameters via DRAM policy data in IAFW/BIOS.

SMIP Definition	Size in Bytes	Description	Default Value
Platform override	4	8 bit value to override the actual Platform ID. 0xff means this is disabled/not used. Any other value than 0xff will be treated as real	0xFF
Reserved	4	Reserved for internal usage	0x00
Message_Level_Mask	4	32 bits used to mask out debug messages.	0x00
PACKAGE	1	0: CoPop/So-Dimm Co-pop if LPDDR3/LPDDR4 adopted and SODIMM if DDR3L is being used	0x00
		1: BGA	
		2: LP3 ACRD	
PROFILE (APL)	1	Note: 1. Only for memory down or downgrade DDR3L frequency 2. LPDDR3/DDR3_freq_X_Y_Z X : tCL; Y: tRCD; Z:tRP 0x1: WIO2_800_7_8_8	0x2A
		0x2: WIO2_1066_9_10_10	
		0x3: LPDDR3_1066_8_10_10	
		0x4: LPDDR3_1333_10_12_12	
		0x5: LPDDR3_1600_12_15_15	
		0x6: LPDDR3_1866_14_17_17	
		0x7: LPDDR3_2133_16_20_20	
		0x8: LPDDR4_1066_10_10_10	
		0x9: LPDDR4_1600_14_15_15	
		0xA: LPDDR4_2133_20_20_20	
		0xB: LPDDR4_2400_24_22_22	
		0xC: LPDDR4_2666_24_24_24	
		0xD: LPDDR4_2933_28_27_27	
		0xE: LPDDR4_3200_28_29_29	
		0xF: DDR3_1066_6_6_6	
		0x10: DDR3_1066_7_7_7	
		0x11: DDR3_1066_8_8_8	
		0x12: DDR3_1333_7_7_7	
		0x13: DDR3_1333_8_8_8	
		0x14: DDR3_1333_9_9_9	
		0x15: DDR3_1333_10_10_10	

SMIP Definition	Size in Bytes	Description	Default Value
		0x16: DDR3_1600_8_8_8 0x17: DDR3_1600_9_9_9	
		0x18: DDR3_1600_10_10_10	
		0x19: DDR3_1600_11_11_11	
		0x1A: DDR3_1866_10_10_10	
		0x1B: DDR3_1866_11_11_11	
		0x1C: DDR3_1866_12_12_12	
		0x1D: DDR3_1866_13_13_13	
		0x1E: DDR3_2133_11_11_11	
		0x1F: DDR3_2133_12_12_12	
		0x20: DDR3_2133_13_13_13	
		0x21: DDR3_2133_14_14_14	
		0x22: DDR4_1333_10_10_10	
		0x23: DDR4_1600_10_10_10	
		0x24: DDR4_1600_11_11_11	
		0x25: DDR4_1600_12_12_12	
		0x26: DDR4_1866_12_12_12	
		0x27: DDR4_1866_13_13_13	
		0x28: DDR4_1866_14_14_14	
		0x29: DDR4_2133_14_14_14	
		0x2A: DDR4_2133_15_15_15	
		0x2B: DDR4_2133_16_16_16	
		0x2C: DDR4_2400_15_15_15	
		0x2D: DDR4_2400_16_16_16	
		0x2E: DDR4_2400_17_17_17	
		0x2F: DDR4_2400_18_18_18	
MemoryDown	1	1: Yes	0x00
		0: No ; It's for populated Dimms (SoDimm and UDimm).	
		2: 1MD+SODIMM (for DDR3L only)	
		3: SODIMM + MD (for DDR3L only)	
		4: 1x32 LPDDR4	
		5: ECC Memory Down	

SMIP Definition	Size in Bytes	Description	Default Value
DDR3LPageSize	1	**NOTE:** Only for memory down 1: 1KB 2: 2KB	0x01
DDR3LASR	1	**NOTE:** Only for memory down 00: Not Supported 01: Supported	0x01
Scrambler Support	1	00: Disabled 01: Enabled	0x01
Channel Hash Mask	2	00: Disabled 01: Enabled	0x00
Slice Hash Mask	2	00: Disabled 01: Enabled	0x00
Interleaved Mode	1	00: Disabled 01: Enabled	0x00
Channels Slices Enable	1	00: Disabled 01: Enabled	0x00
MinRefRate2x Enable	1	00: Disabled 01: Enabled	0x00
Dual Rank Support Enable	1	00: Disabled 01: Enabled	0x01
Disable FastBoot	1	00: Disabled; Used saved training data (if valid) 01: Enabled; Full re-train of memory	0x00
Rmt Mode	1	00: Disabled 03: Enabled All others: Reserved	0x00
RmtCheckRun	1	Parameter used to determine whether to run the margin check. Bit 0 is used for MINIMUM MARGIN CHECK and bit 1 is used for DEGRADE MARGIN CHECK	0x03
RmtMarginChkScale HiThreshld	2	Percentage used to determine the margin tolerances over the failing margin.	0x00C8
Memory Size Limit	2	This value is used to restrict the total amount of memory and the calculations based on it. Value is in MB. Example encodings are: 0x400 = 1GB, 0x800 = 2GB, 0x1000 = 4GB, 0x2000 8GB	0x00
Low Memory Max Value	2	This value is used to restrict the amount of memory below 4GB and the calculations based on it. Value is in MB. Example encodings are:	0x00

SMIP Definition	Size in Bytes	Description	Default Value
		0x400 = 1GB, 0x800 = 2GB, 0x1000 = 4GB, 0x2000 8GB	
High Memory Max Value	2	This value is used to restrict the total amount of memory above 4GB and the calculations based on it. Value is in MB Example encodings are: 0x400 = 1GB, 0x800 = 2GB, 0x1000 = 4GB, 0x2000 8GB	0x00
SPDAddress 0	1	**NOTE:** Only for DDR3L only. Put 0 for MemoryDown. DIMM0 SPD Address	0xA0
SPDAddress 1	1	**NOTE:** Only for DDR3L only. Put 0 for MemoryDown. DIMM1 SPD Address	0xA4
Ch0_RankEnable	1	**NOTE:** Only for memory down [0] Enable Rank 0: Must be set to 1 to enable use of this rank. [1] Enable Rank 1: Must be set to 1 to enable use of this rank.	0x3
Ch0_DeviceWidth	1	**NOTE:** Only for memory down Must specify the DRAM device width per DRAM channel (not to be confused with the SoC Memory Channel width which is always x32 for LPDDR3\LPDDR4 and x64 for DDR3L). LPDDR4 devices typically have two channels per die. 00 - x8 01 - x16 10 - x32 11 - x64	0x2
Ch0_DramDensity	1	**NOTE:** Only for memory down For LPDDR3 and LPDDR4: Must specify the DRAM device density per rank (per Chip Select). The simplest way of identifying the density per rank is to divide the total SoC memory channel density by the number of ranks. For DDR3L: Must specify the DRAM device density per DRAM device. For example, an 8GB 2Rx8 configuration will utilize sixteen 4Gb density DRAMS. In this confirmation, a 4Gb density setting would be selected in the MRC. 000 - 4Gb 001 - 6Gb 010 - 8Gb 011 - 12Gb	0x2

SMIP Definition	Size in Bytes	Description	Default Value
		100 - 16Gb 101 - 2Gb 110-111 - Reserved	
Ch0_Option	1	[0] Rank Select Interleaving Enable. See Address Mapping section for full description. 0 - Rank Select Interleaving disabled 1 - Rank Select Interleaving enabled [1] Bank Address Hashing Enable. See Address Mapping section for full description. 0 - Bank Address Hashing disabled 1 - Bank Address Hashing enabled [3:2] Reserved [5:4] This register specifies the address mapping to be used: 00 - 1KB (A) 01 - 2KB (B)	0x00
Ch0_OdtConfig	1	0 - WEAK_ODT_CONFIG 1 - STRONG_ODT_CONFIG	0x01
Ch0_TristateClk1	1	Parameter used to determine wether to tristate CLK1. Boolean value	0x00
Ch0_Mode2N	1	2N Mode. Boolean value.	0x00
Ch0_OdtLevels	1	Parameter used to determine if ODT will be held high or low 0 - Use MRC default 1 - ODT_AB_HIGH_HIGH	0x00
Ch1_RankEnable	1	**NOTE:** Only for memory down [0] Enable Rank 0: Must be set to 1 to enable use of this rank. [1] Enable Rank 1: Must be set to 1 to enable use of this rank.	0x3
Ch1_DeviceWidth	1	**NOTE:** Only for memory down DRAM Device Data Width populated on Ranks 0 and 1. 00 - x8 01 - x16 10 - x32 11 - x64	0x2

SMIP Definition	Size in Bytes	Description	Default Value
Ch1_DramDensity	1	**NOTE:** Only for memory down DRAM Device Density populated on Ranks 0 and 1. 000 - 4Gb 001 - 6Gb 010 - 8Gb 011 - 12Gb 100 - 16Gb 101-111 – Reserved	0x2
Ch1_Option	1	[0] Rank Select Interleaving Enable. See Address Mapping section for full description. 0 - Rank Select Interleaving disabled 1 - Rank Select Interleaving enabled [1] Bank Address Hashing Enable. See Address Mapping section for full description. 0 - Bank Address Hashing disabled 1 - Bank Address Hashing enabled [3:2] Reserved [5:4] This register specifies the address mapping to be used: 00 - 1KB (A) 01 - 2KB (B)	0x00
Ch1_OdtConfig	1	0 - WEAK_ODT_CONFIG 1 - STRONG_ODT_CONFIG	0x01
Ch1_TristateClk1	1	Parameter used to determine whether to tristate CLK1. Boolean value	0x00
Ch1_Mode2N	1	2N Mode. Boolean value.	0x00
Ch1_OdtLevels	1	Parameter used to determine if ODT will be held high or low 0 - Use MRC default 1 - ODT_AB_HIGH_HIGH	0x00
Ch2_RankEnable	1	**NOTE:** Only for memory down [0] Enable Rank 0: Must be set to 1 to enable use of this rank. [1] Enable Rank 1: Must be set to 1 to enable use of this rank.	0x3
Ch2_DeviceWidth	1	**NOTE:** Only for memory down DRAM Device Data Width populated on Ranks 0 and 1. 00 - x8 01 - x16 10 - x32 11 - x64	0x2
Ch2_DramDensity	1	**NOTE:** Only for memory down DRAM Device Density populated on Ranks 0 and 1. 000 - 4Gb 001 - 6Gb	0x2

SMIP Definition	Size in Bytes	Description	Default Value
		010 - 8Gb 011 - 12Gb 100 - 16Gb 101-111 – Reserved	
Ch2_Option	1	[0] Rank Select Interleaving Enable. See Address Mapping section for full description. 0 - Rank Select Interleaving disabled 1 - Rank Select Interleaving enabled [1] Bank Address Hashing Enable. See Address Mapping section for full description. 0 - Bank Address Hashing disabled 1 - Bank Address Hashing enabled [5:4] This register specifies the address mapping to be used: 00 - 1KB (A) 01 - 2KB (B)	0x00
Ch2_OdtConfig	1	0 - WEAK_ODT_CONFIG 1 - STRONG_ODT_CONFIG	0x01
Ch2_TristateClk1	1	Parameter used to determine whether to tristate CLK1. Boolean value	0x00
Ch2_Mode2N	1	2N Mode. Boolean value.	0x00
Ch2_OdtLevels	1	Parameter used to determine if ODT will be held high or low 0 - Use MRC default 1 - ODT_AB_HIGH_HIGH 3 - ODT_AB_HIGH_LOW	0x00
Ch3_RankEnable	1	**NOTE:** Only for memory down [0] Enable Rank 0: Must be set to 1 to enable use of this rank. [1] Enable Rank 1: Must be set to 1 to enable use of this rank.	0x3
Ch3_DeviceWidth	1	**NOTE:** Only for memory down DRAM Device Data Width populated on Ranks 0 and 1. 00 - x8 01 - x16 10 - x32 11 - x64	0x2
Ch3_DramDensity	1	**NOTE:** Only for memory down DRAM Device Density populated on Ranks 0 and 1. 000 - 4Gb 001 - 6Gb 010 - 8Gb 011 - 12Gb 100 - 16Gb 101-111 – Reserved	0x2

SMIP Definition	Size in Bytes	Description	Default Value
Ch3_Option	1	[0] Rank Select Interleaving Enable. See Address Mapping section for full description. 0 - Rank Select Interleaving disabled 1 - Rank Select Interleaving enabled [1] Bank Address Hashing Enable. See Address Mapping section for full description. 0 - Bank Address Hashing disabled 1 - Bank Address Hashing enabled [3:2] Reserved [5:4] This register specifies the address mapping to be used: 00 - 1KB (A) 01 - 2KB (B)	0x00
Ch3_OdtConfig	1	0 - WEAK_ODT_CONFIG 1 - STRONG_ODT_CONFIG	0x01
Ch3_TristateClk1	1	Parameter used to determine whether to tristate CLK1. Boolean value	0x00
Ch3_Mode2N	1	2N Mode. Boolean value.	0x00
Ch3_OdtLevels	1	Parameter used to determine if ODT will be held high or low 0 - Use MRC default 1 - ODT_AB_HIGH_HIGH 3 - ODT_AB_HIGH_LOW	0x00
Ch0_bit00 to Bit31_Swizzling	32	Channel 0 PHY to Dunit DQ mapping (only used if not 1-1 mapping)	{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}
Ch1_bit00 to Bit31_Swizzling	32	Channel 1 PHY to Dunit DQ mapping (only used if not 1-1 mapping)	{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}
Ch2_bit00 to Bit31_Swizzling	32	Channel 2 PHY to Dunit DQ mapping (only used if not 1-1 mapping)	{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}

SMIP Definition	Size in Bytes	Description	Default Value
Ch3_bit00 to Bit31_Swizzling	32	Channel 3 PHY to Dunit DQ mapping (only used if not 1-1 mapping)	{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}
Profile1	168	Fields (From Field-Package to Field-Ch3_bit00 to Bit31_Swizzling) are repeated for Profile1	Aligned with APL RVP2
Profile2	168	Fields (From Field-Package to Field-Ch3_bit00 to Bit31_Swizzling) are repeated for Profile2	Aligned with APL A0/A1 stepping
Profile3	168	Fields (From Field-Package to Field-Ch3_bit00 to Bit31_Swizzling) are repeated for Profile3	0x00
Profile4	168	Fields (From Field-Package to Field-Ch3_bit00 to Bit31_Swizzling) are repeated for Profile4	0x00
Profile5	168	Fields (From Field-Package to Field-Ch3_bit00 to Bit31_Swizzling) are repeated for Profile5	Aligned with APL B1 stepping

NOTE: No need to set the Chx_bit00 to Bit31_Swizzling for **DDR3L** memory.

6.1.1 Examples for the Memory Configuration

Hybrid Memory design ➔ Channel 0: DDR3L (memory down with 1 Rank) and Channel 1: SO-DIMM

- Set MemoryDown = 2
- Set DIMM SPDAddress[1] = A0 or A4 (based on board design)
- Set Ch0_RankEnable = 1
- Set Ch0_DramDensity = 0 (4Gb) (based on board design)
- Set Ch0_DeviceWidth = 1 (x16) (based on board design)
- Set Ch0_Option = 3 (Rank Select Interleaving enabled & Bank Address Hashing enabled)

6.1.2 CA Mapping

Table 6-1. CA Mapping for All Technologies

CA Mappings	LPDDR3	LPDDR4	DDR3L
0 - uniDIMM/SODIMM	"LP3 Uni-DIMM 2x2Rx32" & "LP3 BGA RVP"	"LP4 - uniDIMM"	"DDR3L SODIMM or BGA or UDIMM Rank0 (non-ECC/ECC)"
1 - BGA	"LP3 BGA PDG" & "LP3 BGA 2x2Rx32 (Chromebook)"	"LP4 BGA 2x2Rx32"	N/A
2 - BGA mirrored (LPDDR3 only)	"LP3 BGA-mirror 2x2Rx32"	N/A	N/A
3 - uniDIMM/SODIMM/UDIMM with Rank 1 Mirrored (DDR3L)	N/A	N/A	"DDR3L (Rank 1 UDIMM)"

Table 6-2. Detail Balls to Pins Mapping

DDRIO2 Ball	DDRIO3 Ball	CA Mapping 0 DDR3L SODIMM or BGA or UDIMM (non-ECC/ECC)	LP3 Uni-DIMM and RVP 2x2Rx32	LP3 BGA PDG	LP3 BGA-mirror 2x2Rx32	LP4 Uni-DIMM 2x2Rx32	LP4 BGA 2x2Rx32
BJ52	BG11	MA5	CAA0	CAA2	CAA7	CAA0	CAA2
BH53	BG12	MA8	CAA3	CAA1	CAA8	CAA3	CAA0
BG53	BJ12	MA6	CAA2	CAA0	CAA9	CAA2	CAA1
BG54	BG13	MA12	CAA6	CAA5	CAA4	NC	NC
BG52	BH11	MA9	CAA1	CAA4	CAA5	CAA1	CAA4

DDRIO2 Ball	DDRIO3 Ball	CA Mapping 0 DDR3L SODIMM or BGA or UDIMM (non-ECC/ECC)	LP3 Uni-DIMM and RVP 2x2Rx32	LP3 BGA PDG	LP3 BGA-mirror 2x2Rx32	LP4 Uni-DIMM 2x2Rx32	LP4 BGA 2x2Rx32
BG55	BG14	MA7	CAA4	CAA3	CAA6	CAA4	CAA3
BE41	BB16	MA4	NC	NC	NC	NC	NC
BG56	BG15	MA14	CAA9	CAA8	CAA1	NC	NC
BH57	BH15	BA2	CAA5	CAA7	CAA2	CAA5	CAA5
BH55	BH13	MA11	CAA7	CAA6	CAA3	NC	NC
BG57	BG16	MA15	CAA8	CAA9	CAA0	NC	NC
BJ48	BH6	BA0	CAB4	CAB2	CAB7	CAB4	CAB4
BG47	BJ6	RASN	CAB3	CAB3	CAB6	CAB3	CAB3
BH51	BH9	MA2	CAB5	CAB5	CAB4	CAB5	CAB5
BG50	BG9	MA0	CAB9	CAB7	CAB2	NC	NC
BH49	BG7	MA10	CAB7	CAB6	CAB3	NC	NC
BG49	BG8	BA1	CAB6	CAB8	CAB1	NC	NC
BD41	BD16	MA3	NC	NC	NC	NC	NC
BG51	BG10	MA1	CAB8	CAB9	CAB0	NC	NC
BH47	BH4	CASN	CAB1	CAB1	CAB8	CAB1	CAB1
BG48	BH7	WEN	CAB2	CAB4	CAB5	CAB2	CAB2
BG46	BH3	MA13	CAB0	CAB0	CAB9	CAB0	CAB0
AR35	AR29	VREFCA	VREFCA	VREFCA	VREFCA	NC	NC
AT34	AT30	VREFDQ	VREFDQ	VREFDQ	VREFDQ	NC	NC
AR34		RESET_n	NC	NC	NC	RESET_n	RESET_n
-	AR30	RESET_n	NC	NC	NC	RESET_n	RESET_n
AV34		RPDEXT	RPDEXT	RPDEXT	RPDEXT	RPDEXT	RPDEXT
	AV30	RPDEXT	RPDEXT	RPDEXT	RPDEXT	RPDEXT	RPDEXT

6.1.3 DQ Mapping for CH0 to CH3

This section defines the connections of all signals between the Dunits the PHYs the package and the DIMMs. PHY byte lanes are not well ordered physically. The DDR3L/LPDDR3 portion of the ball names were meant to untangle the PHY bytes and provide the optimal byte swizzle for each packaging option. However, platforms have not chosen the same swizzle for each packaging option. Below are the DQs for CH0 – CH3. This configuration only applies to LPDDR3 and LPDDR4 technologies.

Table 6-1. DQs for CH0

Channel0

MRC Name	RVP NET Name
DRAM_POLICY.ChSwizzle[0][0]	CH0_DQB8
DRAM_POLICY.ChSwizzle[0] [1]	CH0_DQB9
DRAM_POLICY.ChSwizzle[0] [2]	CH0_DQB10
DRAM_POLICY.ChSwizzle[0] [3]	CH0_DQB11
DRAM_POLICY.ChSwizzle[0] [4]	CH0_DQB12
DRAM_POLICY.ChSwizzle[0] [5]	CH0_DQB13
DRAM_POLICY.ChSwizzle[0] [6]	CH0_DQB14
DRAM_POLICY.ChSwizzle[0] [7]	CH0_DQB15
DRAM_POLICY.ChSwizzle[0] [8]	CH0_DQB0
DRAM_POLICY.ChSwizzle[0] [9]	CH0_DQB1
DRAM_POLICY.ChSwizzle[0] [10]	CH0_DQB2
DRAM_POLICY.ChSwizzle[0] [11]	CH0_DQB3
DRAM_POLICY.ChSwizzle[0] [12]	CH0_DQB4
DRAM_POLICY.ChSwizzle[0] [13]	CH0_DQB5
DRAM_POLICY.ChSwizzle[0] [14]	CH0_DQB6
DRAM_POLICY.ChSwizzle[0] [15]	CH0_DQB7
DRAM_POLICY.ChSwizzle[0][16]	CH0_DQB24
DRAM_POLICY.ChSwizzle[0] [17]	CH0_DQB25
DRAM_POLICY.ChSwizzle[0] [18]	CH0_DQB26
DRAM_POLICY.ChSwizzle[0] [19]	CH0_DQB27
DRAM_POLICY.ChSwizzle[0] [20]	CH0_DQB28
DRAM_POLICY.ChSwizzle[0] [21]	CH0_DQB29
DRAM_POLICY.ChSwizzle[0] [22]	CH0_DQB30
DRAM_POLICY.ChSwizzle[0] [23]	CH0_DQB31
DRAM_POLICY.ChSwizzle[0] [24]	CH0_DQB16
DRAM_POLICY.ChSwizzle[0] [25]	CH0_DQB17
DRAM_POLICY.ChSwizzle[0] [26]	CH0_DQB18
DRAM_POLICY.ChSwizzle[0] [27]	CH0_DQB19
DRAM_POLICY.ChSwizzle[0] [28]	CH0_DQB20
DRAM_POLICY.ChSwizzle[0] [29]	CH0_DQB21
DRAM_POLICY.ChSwizzle[0] [30]	CH0_DQB22
DRAM_POLICY.ChSwizzle[0] [31]	CH0_DQB23

Table 6-2. DQs for CH1

Channel1

MRC Name	RVP NET Name
DRAM_POLICY.ChSwizzle[1][0]	CH0_DQA0
DRAM_POLICY.ChSwizzle[1] [1]	CH0_DQA1
DRAM_POLICY.ChSwizzle[1] [2]	CH0_DQA2
DRAM_POLICY.ChSwizzle[1] [3]	CH0_DQA3
DRAM_POLICY.ChSwizzle[1] [4]	CH0_DQA4
DRAM_POLICY.ChSwizzle[1] [5]	CH0_DQA5
DRAM_POLICY.ChSwizzle[1] [6]	CH0_DQA6
DRAM_POLICY.ChSwizzle[1] [7]	CH0_DQA7
DRAM_POLICY.ChSwizzle[1] [8]	CH0_DQA8
DRAM_POLICY.ChSwizzle[1] [9]	CH0_DQA9
DRAM_POLICY.ChSwizzle[1] [10]	CH0_DQA10
DRAM_POLICY.ChSwizzle[1] [11]	CH0_DQA11
DRAM_POLICY.ChSwizzle[1] [12]	CH0_DQA12
DRAM_POLICY.ChSwizzle[1] [13]	CH0_DQA13
DRAM_POLICY.ChSwizzle[1] [14]	CH0_DQA14
DRAM_POLICY.ChSwizzle[1] [15]	CH0_DQA15
DRAM_POLICY.ChSwizzle[1] [16]	CH0_DQA16
DRAM_POLICY.ChSwizzle[1] [17]	CH0_DQA17
DRAM_POLICY.ChSwizzle[1] [18]	CH0_DQA18
DRAM_POLICY.ChSwizzle[1] [19]	CH0_DQA19
DRAM_POLICY.ChSwizzle[1] [20]	CH0_DQA20
DRAM_POLICY.ChSwizzle[1] [21]	CH0_DQA21
DRAM_POLICY.ChSwizzle[1] [22]	CH0_DQA22
DRAM_POLICY.ChSwizzle[1] [23]	CH0_DQA23
DRAM_POLICY.ChSwizzle[1] [24]	CH0_DQA24
DRAM_POLICY.ChSwizzle[1] [25]	CH0_DQA25
DRAM_POLICY.ChSwizzle[1] [26]	CH0_DQA26
DRAM_POLICY.ChSwizzle[1] [27]	CH0_DQA27
DRAM_POLICY.ChSwizzle[1] [28]	CH0_DQA28
DRAM_POLICY.ChSwizzle[1] [29]	CH0_DQA29
DRAM_POLICY.ChSwizzle[1] [30]	CH0_DQA30
DRAM_POLICY.ChSwizzle[1] [31]	CH0_DQA31

Table 6-3. DQs for CH2

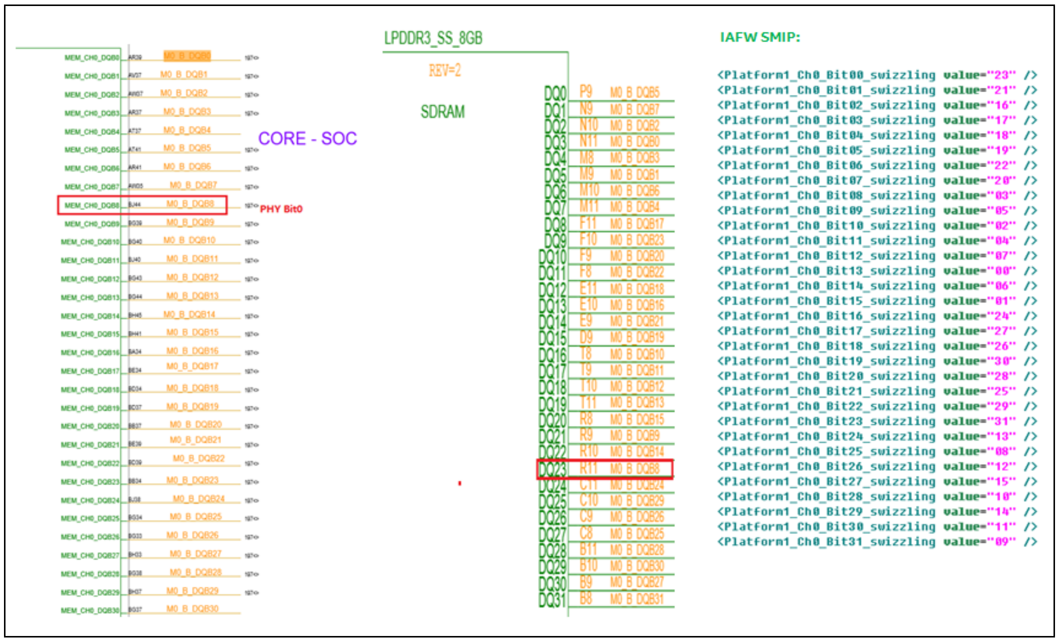
Channel2

MRC Name	RVP NET Name
DRAM_POLICY.ChSwizzle[2][0]	CH1_DQB8
DRAM_POLICY.ChSwizzle[2] [1]	CH1_DQB9
DRAM_POLICY.ChSwizzle[2] [2]	CH1_DQB10
DRAM_POLICY.ChSwizzle[2] [3]	CH1_DQB11
DRAM_POLICY.ChSwizzle[2] [4]	CH1_DQB12
DRAM_POLICY.ChSwizzle[2] [5]	CH1_DQB13
DRAM_POLICY.ChSwizzle[2] [6]	CH1_DQB14
DRAM_POLICY.ChSwizzle[2] [7]	CH1_DQB15
DRAM_POLICY.ChSwizzle[2] [8]	CH1_DQB0
DRAM_POLICY.ChSwizzle[2] [9]	CH1_DQB1
DRAM_POLICY.ChSwizzle[2] [10]	CH1_DQB2
DRAM_POLICY.ChSwizzle[2] [11]	CH1_DQB3
DRAM_POLICY.ChSwizzle[2] [12]	CH1_DQB4
DRAM_POLICY.ChSwizzle[2] [13]	CH1_DQB5
DRAM_POLICY.ChSwizzle[2] [14]	CH1_DQB6
DRAM_POLICY.ChSwizzle[2] [15]	CH1_DQB7
DRAM_POLICY.ChSwizzle[2] [16]	CH1_DQB24
DRAM_POLICY.ChSwizzle[2] [17]	CH1_DQB25
DRAM_POLICY.ChSwizzle[2] [18]	CH1_DQB26
DRAM_POLICY.ChSwizzle[2] [19]	CH1_DQB27
DRAM_POLICY.ChSwizzle[2] [20]	CH1_DQB28
DRAM_POLICY.ChSwizzle[2] [21]	CH1_DQB29
DRAM_POLICY.ChSwizzle[2] [22]	CH1_DQB30
DRAM_POLICY.ChSwizzle[2] [23]	CH1_DQB31
DRAM_POLICY.ChSwizzle[2] [24]	CH1_DQB16
DRAM_POLICY.ChSwizzle[2] [25]	CH1_DQB17
DRAM_POLICY.ChSwizzle[2] [26]	CH1_DQB18
DRAM_POLICY.ChSwizzle[2] [27]	CH1_DQB19
DRAM_POLICY.ChSwizzle[2] [28]	CH1_DQB20
DRAM_POLICY.ChSwizzle[2] [29]	CH1_DQB21
DRAM_POLICY.ChSwizzle[2] [30]	CH1_DQB22
DRAM_POLICY.ChSwizzle[2] [31]	CH1_DQB23

Table 6-4. DQs for CH3

Channel3	
MRC Name	RVP NET Name
DRAM_POLICY.ChSwizzle[3][0]	CH1_DQA0
DRAM_POLICY.ChSwizzle[3] [1]	CH1_DQA1
DRAM_POLICY.ChSwizzle[3] [2]	CH1_DQA2
DRAM_POLICY.ChSwizzle[3] [3]	CH1_DQA3
DRAM_POLICY.ChSwizzle[3] [4]	CH1_DQA4
DRAM_POLICY.ChSwizzle[3] [5]	CH1_DQA5
DRAM_POLICY.ChSwizzle[3] [6]	CH1_DQA6
DRAM_POLICY.ChSwizzle[3] [7]	CH1_DQA7
DRAM_POLICY.ChSwizzle[3] [8]	CH1_DQA8
DRAM_POLICY.ChSwizzle[3] [9]	CH1_DQA9
DRAM_POLICY.ChSwizzle[3] [10]	CH1_DQA10
DRAM_POLICY.ChSwizzle[3] [11]	CH1_DQA11
DRAM_POLICY.ChSwizzle[3] [12]	CH1_DQA12
DRAM_POLICY.ChSwizzle[3] [13]	CH1_DQA13
DRAM_POLICY.ChSwizzle[3] [14]	CH1_DQA14
DRAM_POLICY.ChSwizzle[3] [15]	CH1_DQA15
DRAM_POLICY.ChSwizzle[3] [16]	CH1_DQA16
DRAM_POLICY.ChSwizzle[3] [17]	CH1_DQA17
DRAM_POLICY.ChSwizzle[3] [18]	CH1_DQA18
DRAM_POLICY.ChSwizzle[3] [19]	CH1_DQA19
DRAM_POLICY.ChSwizzle[3] [20]	CH1_DQA20
DRAM_POLICY.ChSwizzle[3] [21]	CH1_DQA21
DRAM_POLICY.ChSwizzle[3] [22]	CH1_DQA22
DRAM_POLICY.ChSwizzle[3] [23]	CH1_DQA23
DRAM_POLICY.ChSwizzle[3] [24]	CH1_DQA24
DRAM_POLICY.ChSwizzle[3] [25]	CH1_DQA25
DRAM_POLICY.ChSwizzle[3] [26]	CH1_DQA26
DRAM_POLICY.ChSwizzle[3] [27]	CH1_DQA27
DRAM_POLICY.ChSwizzle[3] [28]	CH1_DQA28
DRAM_POLICY.ChSwizzle[3] [29]	CH1_DQA29
DRAM_POLICY.ChSwizzle[3] [30]	CH1_DQA30
DRAM_POLICY.ChSwizzle[3] [31]	CH1_DQA31

In the case of DQ mapping for Channel 0 on LPDDR3 RVP, according to the Table 6-2, the PAD name - CH0_DQB8 means PHY 0, Channel 0 and DQ0 (aka ch0_bit 0_Swizzling in SMIP) linked to Pin name -- MEM_CH0_DQB8. Also, based on the board schematics, this pin is connected to the DQ23 of SDRAM. And therefore the DQ mapping for PHY0/CH0 on LPDDR3 RVP is as follow. The same mechanism is applied to the rest of PHYs/Channels.



IAFW SMIP:

```
<Platform1_Ch0_Bit00_swizzling value="23" />
<Platform1_Ch0_Bit01_swizzling value="21" />
<Platform1_Ch0_Bit02_swizzling value="16" />
<Platform1_Ch0_Bit03_swizzling value="17" />
<Platform1_Ch0_Bit04_swizzling value="18" />
<Platform1_Ch0_Bit05_swizzling value="19" />
<Platform1_Ch0_Bit06_swizzling value="22" />
<Platform1_Ch0_Bit07_swizzling value="20" />
<Platform1_Ch0_Bit08_swizzling value="03" />
<Platform1_Ch0_Bit09_swizzling value="05" />
<Platform1_Ch0_Bit10_swizzling value="02" />
<Platform1_Ch0_Bit11_swizzling value="04" />
<Platform1_Ch0_Bit12_swizzling value="07" />
<Platform1_Ch0_Bit13_swizzling value="00" />
<Platform1_Ch0_Bit14_swizzling value="06" />
<Platform1_Ch0_Bit15_swizzling value="01" />
<Platform1_Ch0_Bit16_swizzling value="24" />
<Platform1_Ch0_Bit17_swizzling value="27" />
<Platform1_Ch0_Bit18_swizzling value="26" />
<Platform1_Ch0_Bit19_swizzling value="30" />
<Platform1_Ch0_Bit20_swizzling value="28" />
<Platform1_Ch0_Bit21_swizzling value="25" />
<Platform1_Ch0_Bit22_swizzling value="29" />
<Platform1_Ch0_Bit23_swizzling value="31" />
<Platform1_Ch0_Bit24_swizzling value="13" />
<Platform1_Ch0_Bit25_swizzling value="08" />
<Platform1_Ch0_Bit26_swizzling value="12" />
<Platform1_Ch0_Bit27_swizzling value="15" />
<Platform1_Ch0_Bit28_swizzling value="10" />
<Platform1_Ch0_Bit29_swizzling value="14" />
<Platform1_Ch0_Bit30_swizzling value="11" />
<Platform1_Ch0_Bit31_swizzling value="09" />
```

6.2 IAFW SMIP Footer

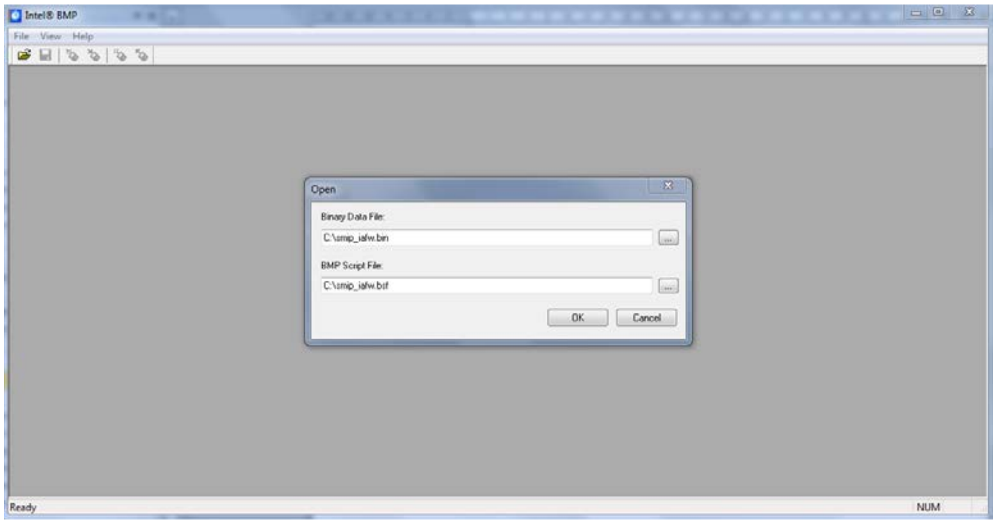
Value: 0xAAFFCCAA.

6.3 IAFW SMIP Modification

The well-Known Intel® BMP (Binary Modification Program) tool is used for configuring IAFW SMIP and the RVP IAFW SMIP.bin and BMP script file (*.bsf) are released along with RVP IAFW/BIOS binary. So, use the BMP to customize the IAFW SMIP and Intel® FIT (Flash Image Tool) included in the TXE Firmware kit for FW ingredient stitch.

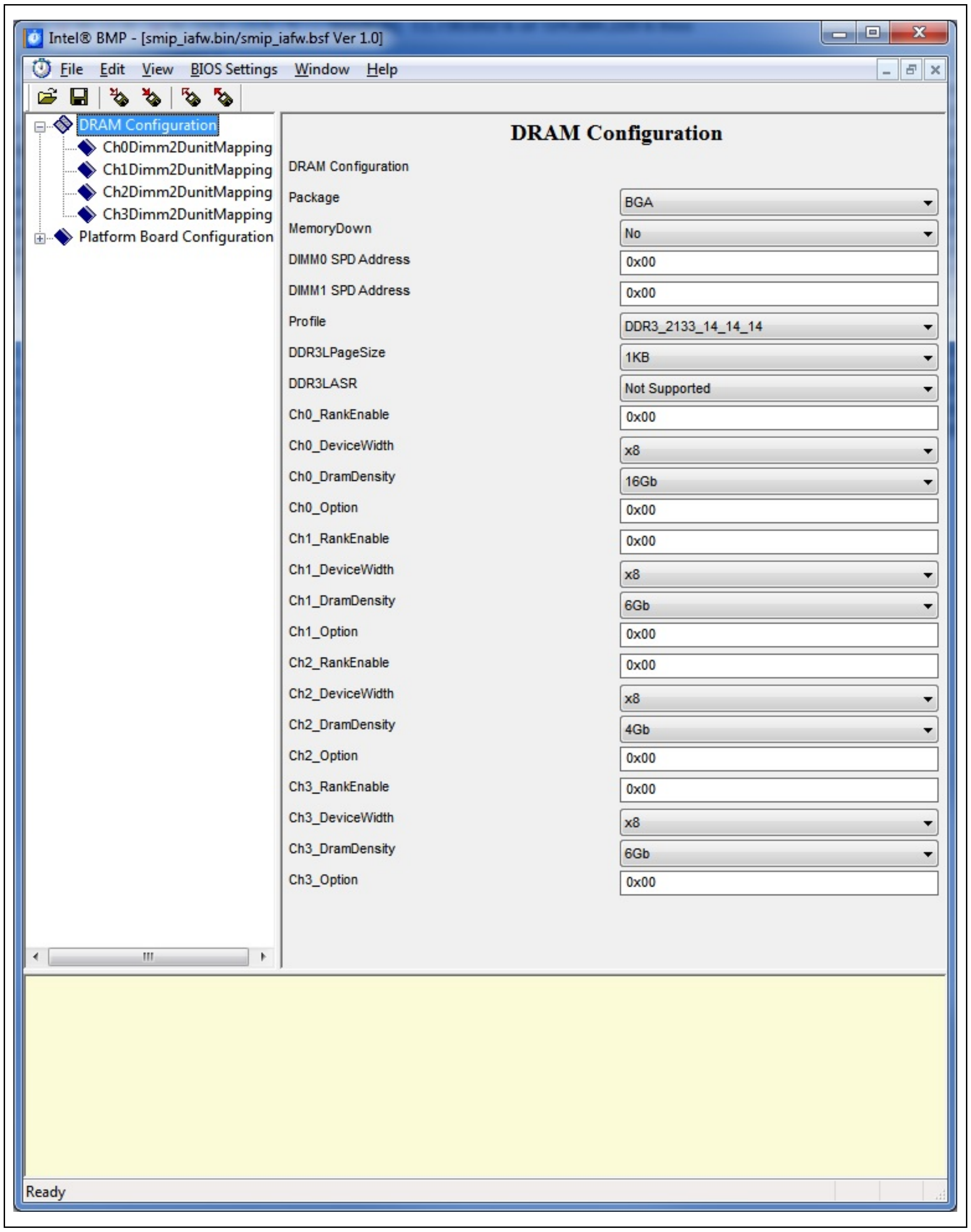
Refer the below steps to customize the IAFW SMIP.

1. Load IAFW SMIP binary and Script file in the RVP IAFW/BIOS binary package.



2. Modify the field based on the board design.

[DRAM Configuration]:


Intel® BMP - [smip_iafw.bin/smip_iafw.bsf Ver 1.0]
File Edit View BIOS Settings Window Help
DRAM Configuration
Ch0Dimm2DunitMapping
Ch1Dimm2DunitMapping
Ch2Dimm2DunitMapping
Ch3Dimm2DunitMapping
Platform Board Configuration
Ready

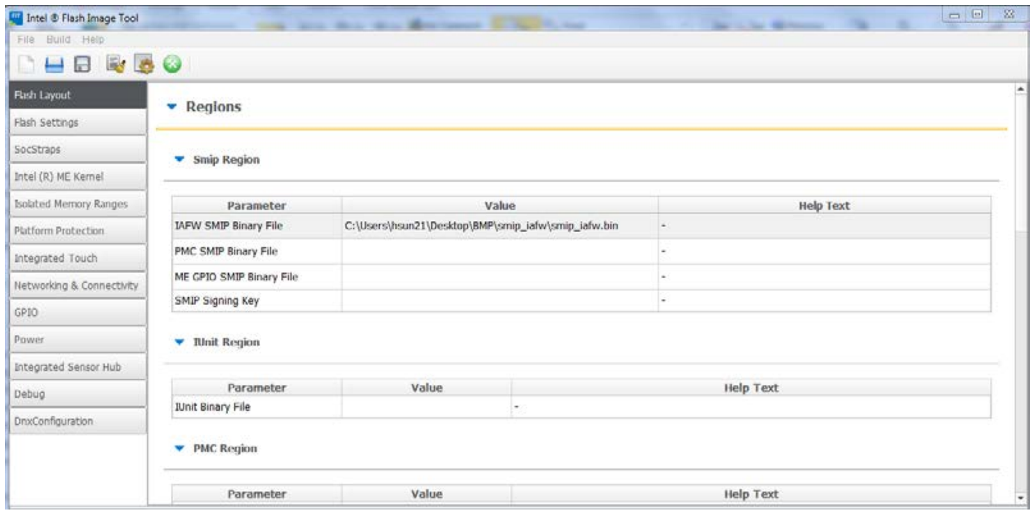
DRAM Configuration

DRAM Configuration

Package	BGA
MemoryDown	No
DIMM0 SPD Address	0x00
DIMM1 SPD Address	0x00
Profile	DDR3_2133_14_14_14
DDR3LPageSize	1KB
DDR3LASR	Not Supported
Ch0_RankEnable	0x00
Ch0_DeviceWidth	x8
Ch0_DramDensity	16Gb
Ch0_Option	0x00
Ch1_RankEnable	0x00
Ch1_DeviceWidth	x8
Ch1_DramDensity	6Gb
Ch1_Option	0x00
Ch2_RankEnable	0x00
Ch2_DeviceWidth	x8
Ch2_DramDensity	4Gb
Ch2_Option	0x00
Ch3_RankEnable	0x00
Ch3_DeviceWidth	x8
Ch3_DramDensity	6Gb
Ch3_Option	0x00

3. Click either “Save” to overwrite the default IAFW SMIP setting or “Save as” to save the customized IAFW SMIP binary.

4. Take IAFW SMIP binary as an input for FIT to create the final IFWI.



Regions

Smip Region

Parameter	Value	Help Text
IAFW SMIP Binary File	C:\Users\hsun21\Desktop\BMP\smip_iafw\smip_iafw.bin	-
PMC SMIP Binary File		-
ME GPIO SMIP Binary File		-
SMIP Signing Key		-

IUnit Region

Parameter	Value	Help Text
IUnit Binary File		-

PMC Region

Parameter	Value	Help Text

§

7 Power Management

The objective of this chapter is to review all power management features and also detail configurations that results in the most energy efficient system. When properly configured yields optimal performance per watts.

7.1 Enhanced Intel® Speedstep Technology

Enhanced Intel® Speedstep technology allows the processor to dynamically adjust frequency and voltage based on power versus performance needs. System BIOS can detect the presence of EIST feature by reading CPUID.(EAX=1):ECX[7]. When set this indicates that this feature is enabled.

The sequence of steps needed to enable Enhanced Intel® Speedstep Technology transitions is outlined as below:

1. Check if all processors support Enhanced Intel® Speedstep Technology.

    a. If the above check for Enhanced Intel® Speedstep Technology support in all processor packages is false, then ensure that Enhanced Intel® Speedstep Technology is disabled on all processor packages by clearing MSR MISC_ENABLES (1A0h) [16] in all processor packages.

    b. If the above check for Enhanced Intel® Speedstep Technology support in all processor packages is true, then set MSR MISC_ENABLES (1A0h) [16]. This register has package-level scope, so the minimum requirement to have one logical processor in each package set this bit.

2. Check if all processors support Burst Mode.

    a. If the above check for Turbo Mode support in all processor packages is false, then ensure that Turbo Mode is disabled on each processor package by setting MSR MISC_ENABLES (1A0h) [38]. This register has package-level scope and is default set to force Turbo as disabled.

    b. If the above check for Turbo Mode support in all processor packages is true then set MSR MISC_ENABLES (1A0h) [38]. This register has package-level scope, so the minimum requirement to have one logical processor in each package set this bit.

3. Ensure that Enhanced Intel® Speedstep Technology Hardware Coordination is enabled. This setting is appropriate for most legacy OS that support Enhanced Intel® Speedstep Technology. However, if the OS indicates support for software coordination during the early ACPI initialization phase, this setting can be changed to disable hardware coordination. To Enable Enhanced Intel® Speedstep Technology Power Management Hardware coordination, clear MSR MISC_PWR_MGMT (1AAh) [0]. This register has package-level scope, so the minimum requirement is to have one logical processor in each package clear this bit to zero.
4. Read the Max Efficiency Ratio from MSR PLATFORM_INFO (CEh) [47:40].
5. Read the Max Non-Turbo Ratio from MSR PLATFORM_INFO (CEh) [15:8].
6. If required, force the system BSP to the Max Non-Turbo Ratio. Program the

Maximum Non-Turbo ratio in MSR IA32_PERF_CTL (199h) [14:8].

7. [Recommended if logical processors will be discovered/managed by OS] Force all processors other than the System BSP to the lowest P-state so that hardware coordination can function as expected. For each logical processor in each package, program the Max Efficiency Ratio into the MSR IA32_PERF_CTRL (199h) [14:8].
8. IA32_PERF_CTRL has thread-level scope, so this step must be performed on all logical processor (excluding the system BSP). This is also referred to as Max efficiency boot.
9. Compute the set of valid operating points or P-states.
10. Build the ACPI structures required to report Enhanced Intel® Speedstep Technology support to the OSPM.

7.1.1 Determining Number of Operating Points

The number of operating points varies, and is based on the difference between the maximum and minimum bus ratios and whether Turbo mode is available and enabled. All ratios between the minimum ratio and maximum ratio may be used as valid P-state ratios. BIOS may choose 1 or more as a valid P-state ratio step size. For most optimized results from the Operating system perspective, it is recommended that system BIOS limit the number of P states to 16.

Example 7-1. Number of States Calculation

```
MaxNonTurboRatio = PLATFORM_INFO bits [15:8]

MaxEfficiencyRatio = PLATFORM_INFO bits [47:40]

RatioRange = MaxNonTurboRatio - MaxEfficiencyRatio

RatioStepSize = 1

NumStates=Floor((RatioRange / RatioStepSize)   + 1

If Turbo Mode is available and enabled

Then NumStates = NumStates + 1

If (NumStates > 16)

Then RatioStepSize = RatioStepSize + 1

And NumStates = Floor((RatioRange / RatioStepSize) + 1
```

If a Ratio Step Size greater than 1 is used, the FLOOR operation may result in processors with odd maximum bus ratios having the bus ratio step between P[0] and P[1] greater than the ratio step size.

The ACPI specification strictly recommends no more than 16 Operating points for optimum performance. It is hence recommended that the step size be increased in case the number of operating points exceeds 16.

7.1.2 Minimum and “Low Frequency Mode” Bus Ratio

Starting from Cedarview processor, APL SoC distinguishes the Minimum Bus Ratio and the “Low Frequency Mode” (LFM) Bus Ratio. The MIN_RATIO is the lowest processor core clock to reference bus clock ratio supported, which corresponds to its minimum performance state. The LFM_RATIO is the highest processor core clock to reference bus clock ratio supported at lowest core operating voltage (Vcc). This is the frequency at which the APL SoC will boot at.

7.1.3 Calculating Enhanced Intel® Speedstep Technology

Intermediate Points

The example pseudocode below illustrates how the intermediate performance states can be evaluated for best power versus performance benefits keeping in mind the ACPI spec recommendations.

```
// Fill the 16 states in P-state array

NumGaps = NumGaps - 1

If Turbo Mode is available and enabled

{

// turbo ratio is reported directly from processor reported value

pstate_ratio[0] = MSR 1ADh [7:0]

Then NumGaps = NumGaps - 1 // account for actual number of P states

for(i = 1;i < NumStates; i++)

{

pstate_ratio[i] = MaxNonturboRatio - ((((i-1) * RatioRange)+(NumGaps/2))/

NumGaps

}

}

else

{

for(i = 0;i < NumStates; i++)

{

pstate_ratio[i] = MaxNonturboRatio - (((i * RatioRange)+(NumGaps/2))/

NumGaps

}

}
```

7.1.4 Calculating Power Estimates

Thermal design power values are factory-configured and are published in the processor's EMTS. Intermediate power numbers can be calculated using the algorithms noted below, where n indicates intermediate states ranging from P[NumStates - 1] to P[1].

1. Read the TDP using the value from MSR PKG_POWER_LIMIT (610) [14:0] and the Energy unit value from MSR PKG_POWER_SKU_UNIT (606h) [15:8] to convert the TDP value to Watts.
2. Read the factory-configured Max Non-Turbo Ratio. If the factory-configured Maximum Non-Turbo Ratio has not been modified using the MSR FLEX_RATIO (194h), then Maximum Non-Turbo Ratio is equal to MSR PLATFORM_INFO (CEh) [15:8]. If the Maximum Non-Turbo Ratio has been modified using the MSR FLEX_RATIO (194h), then MSR PLATFORM_INFO (CEh) [15:8] no longer represent the factory-configured Maximum Non-Turbo Ratio and BIOS can either (1) On production units parse the Brand String returned by the CPUID instruction since it always contains the original frequency or (2) Save the original Max Non-Turbo ratio in nonvolatile storage before using the MSR FLEX_RATIO (194h).
3. Compute the P-state for power for each Non-Turbo Ratio using the formula

   M = ((1.1 - Max_Nonturbo_ratio - ratio) * 0.00625)) / 1.1) ^ 2

   P_state_power = ((Ratio/Max_Nonturbo_ratio)) * M * TDP
4. Max turbo state power is the TDP value.

The resulting power numbers may be rounded. They are not expected to exactly match the power used by the processor running at the specified operating point, but provide a relative scaling for OSPM policy to rely on, if desired.

7.1.5 Building ACPI Structures

Required ACPI structures and methods for APL family processors with the Enhanced Intel® Speedstep Technology feature. For details on each of these methods refer ACPI specification.

1. _OSC and _PDC Methods
2. _PSS Method
3. _PCT Object
4. _PPC Method
5. _PSD Object

_OSC (Operating System Details) and _PDC (Processor Driver Capabilities)

_OSC and _PDC Methods are both required. The _OSC and _PDC methods are used to confirm the OSPM capabilities including the support for MP Enhanced Intel® Speedstep Technology and whether OSPM software coordination is available. Call the _OSC method from the _PDC method to prevent duplication. Refer the ACPI specification for background details on these ACPI methods. The reference document for the

Intel-specific _PDC definitions is the Intel Processor Vendor-Specific ACPI: Interface Specification, which is available online at http://download.intel.com/technology/IAPC/acpi/downloads/.

The _PDC flags that impact the IA FW configuration of Enhanced Intel® Speedstep Technology related ACPI structures are listed in Table 7-2.

Table 7-1. _PDC Bit Definitions for P-States

Bit [0]	OSPM is capable of direct access to Performance State MSR.
Bit [1]	OSPM supports the C1 "I/O then Halt" FFH sequence for multi-processor.
Bit [2]	OSPM is capable of direct access to On Demand throttling MSR's.
Bit [3]	OSPM is capable of independent C1, P state and T state support for each processor for multi-processor configurations. This does not indicate support for asymmetrical _CST, _PSS, or _TSS objects for individual processors in the system.
Bit [4]	OSPM is capable of independent C2 and C3 state support for each processor for multi-processor configurations. This bit does not indicate support for asymmetrical _CST objects for individual processors in the system.
Bit [5]	OSPM is capable of native P State software coordination using IA FW supplied _PSD information for multi-processor configurations. OSPM is capable of software coordination of P States, i.e., OSPM supports use of _PSD (SW_ANY and SW_ALL).
Bit [6]	OSPM is capable of native C State software coordination using IA FW supplied _CSD information for multi-processor configurations.
Bit [7]	OSPM is capable of native ACPI throttling software coordination using IA FW supplied _TSD information for multi-processor configurations.
Bit [8]	OSPM is capable of performing native C State instructions (beyond halt) for the C1 handler in multi-processor configurations.
Bit [9]	OSPM is capable of performing native C State instructions for the C2/C3 handler in multi-processor configurations.
Bit [10]	Reserved.
Bit [11]	OSPM is capable of hardware coordination awareness of P states using accounting mechanisms. OSPM will track hardware coordination of P states (HW_ALL) using ACNT and MCNT.
0x0c-0x1f	Reserved – must be reset to zero.

In addition, if the OSPM is ACPI 3.0 compliant, the _PDC bits can also be used to determine whether hardware coordination should remain in the enabled state (which is the default IA FW behavior) and also to expose the hardware dependencies which may exist between logical processors. Refer Table 7-3 for the _PDC bits that impact the _PSD object and hardware coordination.
.

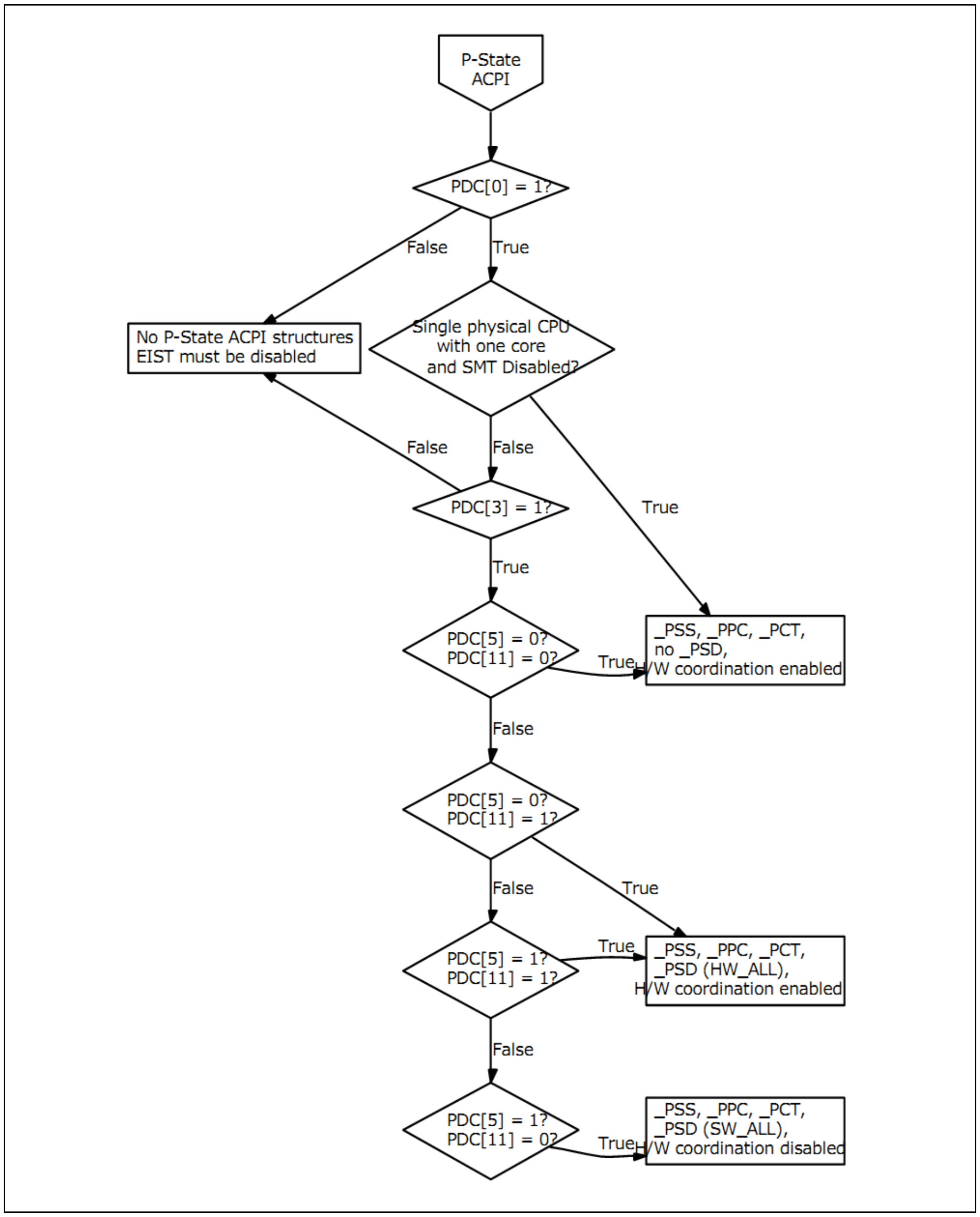
Table 7-2. Interpreting _PDC Parameters for _PSD and Coordination Type

PDC [5] (PSD)	PDC [11] (ACNT/MCNT)	Coordination Type for PSD	Comment
0	0	PSD Required Use	HW coordination.

PDC [5] (PSD)	PDC [11] (ACNT/MCNT)	Coordination Type for PSD	Comment
		HW_ALL	
0	1	PSD Required Use HW_ALL	HW coordination with OS accounting
1	0	PSD Required Use SW_ALL or SW_ANY	Full SW coordination by OSPM
1	1	PSD Required Use HW_ALL	HW coordination with OS accounting

A flowchart of the flow for interpretation of the _PDC parameters exists in Figure 7-1. This flowchart describes the IA FW steps which must occur in the _PDC method to determine which ACPI methods and objects should be exposed to the OSPM.

Figure 7-1. Interpreting _PDC Parameters to Expose ACPI Structures


P-State ACPI
PDC[0] = 1?
False
True
No P-State ACPI structures EIST must be disabled
Single physical CPU with one core and SMT Disabled?
False
False
True
PDC[3] = 1?
True
PDC[5] = 0? PDC[11] = 0?
True
_PSS, _PPC, _PCT, no _PSD, H/W coordination enabled
False
PDC[5] = 0? PDC[11] = 1?
False
True
PDC[5] = 1? PDC[11] = 1?
True
_PSS, _PPC, _PCT, _PSD (HW_ALL), H/W coordination enabled
False
PDC[5] = 1? PDC[11] = 0?
True
_PSS, _PPC, _PCT, _PSD (SW_ALL), H/W coordination disabled

_PSS (Performance Supported States)

This optional object indicates to OSPM the number of supported processor performance states that any given system can support. This object evaluates to a packaged list of information about available performance states including internal CPU core frequency, typical power dissipation and control register values needed to transition between performance states, and status register values that allow OSPM to verify performance transition status after any OS-initiated transition change request. The list is sorted in descending order by typical power dissipation. As a result, the zeroth entry describes the highest performance state and the 'nth' entry describes the lowest performance state.

For APL SoC, the _PSS object should report the performance states computed for each processor. Refer Table 7-3 for details on computing the P-state operating points and the associated data.

When Burst Mode is unavailable or disabled, the operating points range from the Max Non-Burst Ratio to the Max Efficiency Ratio. The core frequency is determined by multiplying the target ratio by the 100 MHz BCLK frequency (use 100 for computations). The control and status field included in each P-state is computed based on each target ratio. The relative power can be estimated using the algorithms.

When Burst Mode is available and enabled, the highest P-state P(0) represents Burst Mode. For this P-state, the core frequency should be listed as Max Non-Burst frequency + 1 MHz and the control/status values should be listed as the Max Burst Ratio. The remaining fields of the Burst Mode P-state should match P(1) which is based on the Max Non Burst Ratio. The remaining operating points range from the Max Non-Burst Ratio to the Max Efficiency Ratio. The Ratio included in each P-state is computed based on each target ratio.

Table 7-3. Composing P-States for _PSS

_PSS P-State Field	P(0) when Burst is available and Enabled	P(0) when Burst is Unavailable or Disabled	All other P-states
Core Frequency	(Max Non-Burst Ratio * BUS Frequency MHz) + 1 MHz **NOTE:** The frequency listed for the Burst P-state is not intended to indicate the actual Burst frequency. Instead it represents the minimum frequency that an OS can expect in this P-state. Since Burst activation is opportunistic based on operating conditions, the frequency listed for Burst is just 1MHz	(Max Non-Burst Ratio * BUS Frequency MHz)	Target Ratio * BUS Frequency MHz
Power	Equivalent to power computed for Max Non-Burst Ratio.	Power computed for Max Non-Burst Ratio.	Relative power computed based on the target ratio.
Transition Latency	10	10	10
Bus Master Latency	10	10	10
Control	Max Burst Ratio, MaxBurstVid	Max Non-Burst Ratio, Max Non-Burst Vid	Target Ratio, Target Vid

_PSS P-State Field	P(0) when Burst is available and Enabled	P(0) when Burst is Unavailable or Disabled	All other P-states
Status	Max Burst Ratio, MaxBurstVid	Max Non-Burst Ratio, Max Non-Burst Vid	Target Ratio, Target Vid

Example 7-2. _PSS Listing (Burst Unavailable or Disabled)

```
Assumptions:
       Burst Mode is Unavailable or Disabled) Max Non-Burst Ratio = 10h
  Max Efficiency Ratio = 0Ch
       Ratio Step Size = 1

Name(_PSS, Package()
{
       Package()  // Actual P[0] State

       { 1600, TBD, 10, 10, 0x0010, 0x0010 }, // The ctl and status fields
should be defined as 0xRRVV, where the upper 8-bits are the BusRatio and
lower 8-bits are VID, determined by IACORE_RATIO/VIDS MSR or interpolation
  Package()      // Actual P[1] State

  { 1500, TBD, 10, 10, 0x000F, 0x000F },

  Package()      // Actual P[2] State
       { 1400, TBD, 10, 10, 0x000E, 0x000E },

       Package()  // Actual P[3] State
  { 1300, TBD, 10, 10, 0x000D, 0x000D },

  Package()      // Actual P[4] State
       { 1200, TBD, 10, 10, 0x000C, 0x000C },
```

Example 7-3. _PSS Listing (Burst is Available and Enabled)

```
Assumptions:
      Burst is available and enabled
  Max Burst Ratio = 14h
      Max Non-Burst Ratio = 10h Max Efficiency Ratio = 0Ch Ratio Step Size
= 1

Name(_PSS, Package()
{
      Package()  // Actual P[0] Burst Mode State
  { 1601, TBD, 10, 10, 0x0014, 0x0014 }, // same comment as previous box.
Burst Ratio/VID should come from FREQ_LIMIT_RATIOS/VIDS.

  Package()      // Actual P[1] State
      { 1600, TBD, 10, 10, 0x0010, 0x0010 },

      Package()  // Actual P[2] State
  { 1500, TBD, 10, 10, 0x000F, 0x000F },

  Package()      // Actual P[3] State
      { 1400, TBD, 10, 10, 0x000E, 0x000E },

      Package()  // Actual P[4] State
  { 1300, TBD, 10, 10, 0x000D, 0x000D },

  Package()      // Actual P[5] State
      { 1200, TBD, 10, 10, 0x000C, 0x000C },
}
```

_PCT (Performance Control)

This object declares the interface that allows OSPM to transition the processor into a performance state. OSPM performs processor performance transitions by writing the performance state–specific control value to a Performance Control Register (refer EDS for details on the IA32_PERF_CTRL MSR).

OSPM may select a processor performance state as indicated by the performance state value returned by the _PPC method, or any lower power (higher numbered) state. The control value to write is contained in the corresponding _PSS entry's "Control" field. Success or failure of the processor performance transition is determined by reading a Performance Status Register (refer EDS for details on the IA32_PERF_STS MSR) to determine the processor's current performance state. If the transition was successful, the value read from PERF_STATUS will match the "Status" field in the _PSS entry that corresponds to the desired processor performance state.

For APL SoC, the _PCT method should always report native mode which is encoded as Functional Fixed Hardware (i.e., FFH). Refer below example for an example of the _PCT method.

Example 7-4. _PCT Listing

```
Assumptions:
      Native mode is always used for Apollo Lake SoC.

Method(_PCT,0)
{
      Return(Package()
  {
       ResourceTemplate(){Register(FFixedHW, 0, 0, 0)},
    ResourceTemplate(){Register(FFixedHW, 0, 0, 0)}
      }
}
```

_PPC (Performance Present Capabilities)

This optional object is a method that dynamically indicates to OSPM the number of performance states currently supported by the platform. This method returns a number that indicates the _PSS entry number of the highest performance state that OSPM can use at a given time.

Example 7-5. _PPC Listing

```
Assumptions:
      All P-states are available for use by OSPM

Method(_PPC,0)
{
      Return(ZERO)   // Return All States Available.
}
```

_PSD (P-State Dependency)

This optional object provides P-state control cross logical processor dependency information to OSPM. The _PSD object evaluates to a packaged list of information that correlates with the P-state information returned by the _PSS object. Each packaged list entry identifies a dependency domain number for the logical processor's P-states, the coordination type for that P-state and the number of logical processors belonging to the domain. Refer the ACPI Specification Revision 3.0b for full details on _PSD object.

A _PSD object should exist in each processor’s scope. The following _PSD object fields must be determined based on the processor topology and the number of processor packages: Domain, Coordination Type, and Number of Processors in the Domain.

The Domain is defined as the dependency domain number to which this P-state entry belongs. The domain for all logical processors in a specific processor package must be the same, indicating to the OSPM that all the processors use the same VR and all transition to the same P-state. A system with one physical package will have one _PSD domain. Systems with 2 physical processors have two _PSD domains. An MP system with multiple processors will have one domain for each physical package, regardless of how many logical processors in each physical package.

The Coordination Type is defined as the type of coordination that exists (hardware) or is required (software) as a result of the underlying hardware dependency. Refer Section 0 for details on P-state coordination on APL SoC. The coordination type could be either 0xFC (SW_ALL), 0xFD (SW_ANY) or 0xFE (HW_ALL) indicating whether OSPM is responsible for coordinating the P-state transitions among processors with dependencies (and needs to initiate the transition on all or any processor in the domain) or whether the hardware will perform this coordination.

Number of Processors in this Domain is defined as the number of processors belonging to the domain for this logical processor's P-states. OSPM will not start performing power state transitions to a particular P-state until this number of processors belonging to the same domain have been detected and started.

For APL SoC, all logical processors in the same processor package are in the same domain. So the number of processors in a specific domain equals the number of processors returned by CPUID.(EAX=0B, ECX=1):EBX[15:0] for physical processor.

Refer Section 7.6.4.1 to review how the _PDC parameter bits are interpreted to determine whether hardware coordination remains enabled (which is the IA FW default) or whether hardware coordination must be disabled during ACPI initialization.

Refer below table for how to set HW_ALL, SW_ALL or SW_ANY for APL SoC:

SINGLE _PCTL	INDP_AU TOCM	_PSD domain	Pstate Coordination	Example PSD Table

0	0	Per Package	HW_ALL(using MIN_CLIP)	// SINGLE_PCTL=0 implies HW_ALL coordination. EHalt disabled //so all cores in same domain.
				// Punit will force all cores to the max resolved frequency //through MIN_CLIP
				
				Scope(_PR.CPU1) // Package 0 Core 0
				{
				//{#entries, revision, domain#, HW_ALL, #cpus_in_domain}
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}
				Scope(_PR.CPU1) // Package 0 Core 1
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}
				Scope(_PR.CPU2) // Package 0 Core 2
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}

SINGLE _PCTL	INDP_AUTOCM	_PSD domain	Pstate Coordination	Example PSD Table

0	1	Per Core	HW_ALL	// EHalt is enabled so each core is in its own domain.
				// Each core can operate at a different virtual frequency.
				// Punit must NOT do any cross-module MIN_CLIP enforcement.
				
				Scope(_PR.CPU1) // Package 0 Core 0
				{
				// {#entries, revision, domain#, HW_ALL, #cpus_in_domain}
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}
				Scope(_PR.CPU1) // Package 0 Core 1
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}
				Scope(_PR.CPU2) // Package 0 Core 2
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFE, 2} })
				}

SINGLE_PCTL	INDP_AUTOCM	_PSD domain	Pstate Coordination	Example PSD Table

1	Do not-care	Per Package	SW_ALL	//Assuming Microcode SINGLE_PCTL BUG NOT FIXED.
				// EHalt should not be set if SINGLE_PCTL=1.
				// Each module is a domain.
				// OS Must do SW_ALL coordination between Modules to get V^2 //power savings. Otherwise will get >=linear savings.
				
				Scope(_PR.CPU1) // Package 0 Core 0
				{
				// {#entries, revision, domain#, SW_ALL, #cpus_in_domain}
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFC, 2} }) }
				Scope(_PR.CPU1) // Package 0 Core 1
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFC, 2} })
				}
				Scope(_PR.CPU2) // Package 0 Core 2
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFC, 2} })
				}

SINGLE _PCTL	INDP_AU TOCM	_PSD domain	Pstate Coordination	Example PSD Table

1	Do not-care	Per Package	SW_ANY	// EHalt should not be set if SINGLE_PCTL=1.
				// All cores in the same domain.
				//Microcode (OR IA FW) must write a bit in Punit when //SINGLE_PCTL=1.
				// Punit must update the MIN_GV_CLIP and MAX_RATIO_LIMIT //to the requested ratio whenever ANY module requests a GV //transition.
				
				Scope(_PR.CPU1) // Package 0 Core 0
				{
				// {#entries, revision, domain#, SW_ANY, #cpus_in_domain}
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFD, 2} }) }
				Scope(_PR.CPU1) // Package 0 Core 1
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFD, 2} })
				}
				Scope(_PR.CPU2) // Package 0 Core 2
				{
				Name (_PSD, Package() { Package(){5, 0, 0, 0xFD, 2} })
				}

P-State Coordination

There are three models of P-state resolution defined by ACPI: SW_ANY, SW_ALL, and HW_ALL. APL SoC has specific support for SW_ANY, but do not prevent use of SW_ALL. The selection of a specific P-state coordination type (e.g., HW_ALL, SW_ALL or SW_ANY) is how the IA FW communicates the P-state support model to the Operating System Power Management (OSPM) via the ACPI _PSD structures. Refer Section 0 for details on how the OSPM uses the _PSD parameters to indicate the possible P-state coordination types supported. Refer Table 7-3 for the _PDC bits that impact the _PSD object and hardware coordination.

7.1.6 SW_ANY

In SW_ANY mode, the OS Power Manager (OSPM) is responsible for coordinating the P-state among logical processors with dependencies (all Logical Processors in a package) and may initiate the transition on any of those Logical Processors in the domain. The expectation from the OSPM is that transitions initiated on any logical processor in a specific domain will force a transition on all logical processors in that domain. Thus the OSPM is only required to initiate a single transition per processor domain. Disabling Enhanced Intel® Speedstep Technology Hardware Coordination forces the processor into SW_ANY mode. SW_ANY should not be used unless the OS explicitly indicates support for this model.

If SW_ANY is used then IA FW must:

- IA FW must build and include an ACPI _PSD object in each processor's scope to provide details on the dependencies between logical processors in each domain, specifically indicating that all logical processors in a physical package are in a single domain and must transition lockstep with each other. Additionally each _PSD must use SW_ANY as the coordination type.

In SW_ANY mode, the OSPM must track a virtual P-state for each logical processor in each domain. And based on these virtual P-states, the OPSM internally calculates the highest performance ratio from all of the requested ratios in a package and handles forcing all logical processors to that P-state. And since hardware coordination is disabled, the OPSM can write this calculated P-state to ANY logical processor in the package to force all logical processors in the package to the desired P-state.

7.1.7 SW_ALL

In SW_ALL mode, the OSPM is responsible for coordinating the P-state among logical processors with dependencies (all logical processors logical processors in a physical package) and must initiate the transition on all of those Logical Processors. The expectation from the OSPM is that transitions must be initiated on every logical processor in a specific domain in order to force a transition on all logical processors in that domain. Thus the OSPM is required to initiate transitions on all processors in a specific domain before any will transition. No specific support exists to force a SW_ALL mode in the APL SoC. Without the Enhanced Intel® Speedstep Technology Hardware Coordination in APL SoC, SW_ANY mode is always set. And the processor in SW_ANY mode can also support SW_ALL.

SW_ALL should not be used unless the OS does not support both SW_ANY mode and HW_ALL mode.

If SW_ALL is used then IA FW must:

- IA FW must build and include an ACPI _PSD object in each processor's scope to provide details on the dependencies between logical processors in each domain, specifically indicating that all logical processors in a physical package are in a single domain and must transition lockstep with each other. Additionally each _PSD must use SW_ALL as the coordination type.

In SW_ALL mode, the OSPM must track a virtual P-state for each logical processor in each domain. And based on these virtual P-states, the OPSM internally calculates the highest performance ratio from all of the requested ratios in a package and handles forcing all logical processors to that P-state. And since hardware coordination is disabled, the OPSM must write this calculated P-state to ALL logical processors in the package to force all logical processors in the package to the desired P-state.

7.1.8 HW_ALL

In HW_ALL mode, the processor hardware is responsible for coordinating the P-state among logical processors with dependencies (all logical processors in a package). The OS is responsible for keeping the P-state requests up to date on all logical processors.

If HW_ALL is used then IA FW must:

- IA FW must program PMG_CST_CONFIG_CONTROL (MSR E2h) Bit 0 = 0 to enable Enhanced Intel® Speedstep Technology Hardware Coordination
- IA FW must build and include an ACPI _PSD object in each processor's scope to provide details on the dependencies between logical processors in each domain, specifically indicating that all logical processors in a physical package are in a single domain and must transition lockstep with each other. Additionally each _PSD must use HW_ALL as the coordination type.

In HW_ALL mode, hardware coordination of P-states is enabled, so the OSPM is not required to track virtual P-state for each core, processor hardware does that. However, the OSPM may optionally use the ACNT and MCNT registers to perform hardware accounting of the P-states for each logical processor in each domain. And based on this accounting feedback, the OPSM internally calculates and optimizes the appropriate performance ratio from all of the requested ratios in a package and handles forcing all logical processors to that P-state. And since hardware coordination is enabled, the OPSM must write this calculated P-state to ALL logical processors in the package to force all logical processors in the package to the desired P-state.

7.1.9 Time Stamp Counter Behavior

The Time Stamp counter (TSC) on the APL SoC is designed so that from a software perspective it appears to run at the maximum frequency of the specific processor (i.e., for a 3 GHz processor, it runs at 3 GHz) independent of the processor performance state that the processor is operating at. The TSC continues to run at the maximum processor frequency throughout a frequency transition resulting from an Enhanced Intel SpeedStep Technology request. However, the Time Stamp Counter will still stop upon entry into either the Deep Sleep or Deeper Sleep states. ACPI compliant operating systems are expected to use a system timer (such as the ACPI timer or High Performance Event Timer) that is not impacted by the processor power savings or C-States and are expected to be immune to this behavior of the time stamp counter.

Also, all 64 bits of the Time Stamp counter are writeable (which is different from previous Intel IA32 processors in which only the lower 32 bits of the Time Stamp counter were writeable – writing of the lower 32 bits would clear the upper 32 bits). If using the Time Stamp counter, software must ensure the use of all 64 bits.

7.2 Burst Mode

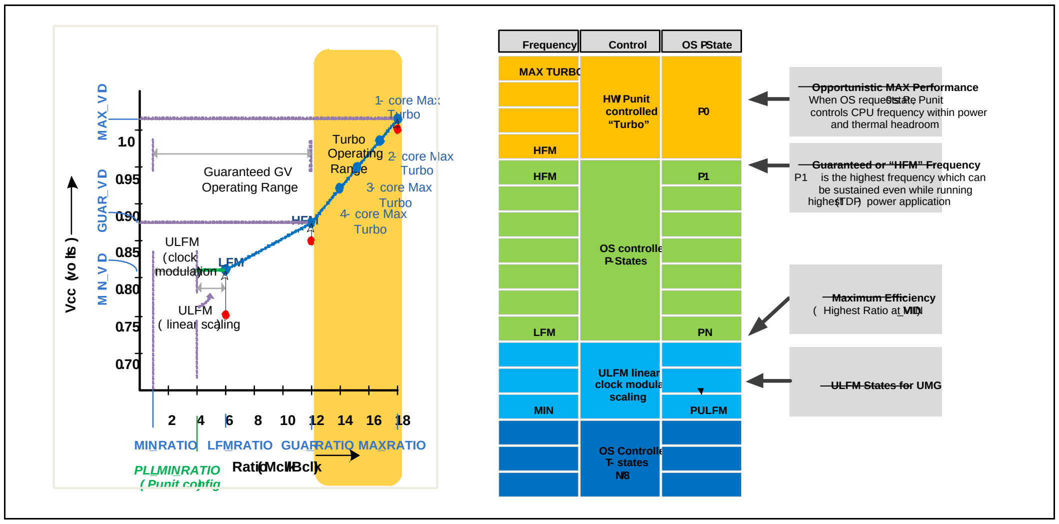
Burst Mode opportunistically, and automatically, allows processor cores to run faster than the marked frequency if the physical processor is operating below power, temperature and current specification limits. Burst Mode is not limited to only a single core or logical processor.

APL SoC Burst Mode can also dynamically utilize the thermal capacity associated with the package and run at higher power levels than the sustained TDP power for very short durations to improve system responsiveness on short and bursty workload conditions.

CPU Burst Mode can be enabled or disabled by IA FW. But Burst Mode operates under OS control by only engaging when OS requests the highest Enhanced Intel® Speedstep Technology performance state (P0). No operating system changes are required to support Burst Mode.

Burst Mode availability is independent of the number of active cores; however, the maximum Burst Mode frequency is dependent on the number of active cores and varies based on the specific factory configuration. Amount of time the system spends in Burst Mode will depend on workload, operating environment and system design.

Figure 7-2. Burst Overview


MAX_VD
GUAR_VD
MIN_VD
Vcc (volts)
1.0
0.95
0.90
0.85
0.80
0.75
0.70
Guaranteed GV Operating Range
Turbo Operating Range
1-core Max Turbo
2-core Max Turbo
3-core Max Turbo
4-core Max Turbo
HFM
LFM
ULFM (clock modulation)
ULFM (linear scaling)
2 4 6 8 10 12 14 16 18
MINRATIO LFMRATIO GUARRATIO MAXRATIO
PLLMINRATIO (Punit config)
Ratio (MCLK/BCLK)
Frequency
Control
OS PState
MAX TURBO
HFM
HFM
LFM
MIN
HW Punit controlled "Turbo"
OS controlled P-States
ULFM linear clock modulation scaling
OS Controlled T-states N/8
P0
P1
PN
PULFM
Opportunistic MAX Performance
When OS requests P0 state, Punit controls CPU frequency within power and thermal headroom
Guaranteed or "HFM" Frequency
P1 is the highest frequency which can be sustained even while running highest (TDP) power application
Maximum Efficiency
(Highest Ratio at MIN)
ULFM States for UMG

7.2.1 Burst Mode Overview

Burst Mode feature is available on some processor configurations. The Burst Mode capability is designed to allow the processor to assess its own thermals, current and power to come up with a dynamic upper limit on its frequency benefit.

On processors with Burst Mode Capability, the processor has the ability to run at frequencies higher than the advertised frequency of the part when requested and when operating conditions allow higher frequency.

In Burst Mode, the processor could be running any ratio in the inclusive range between Max Burst and Max Non-Burst. For most processors, the Max Burst Ratio is factory-configured and visible to IA FW via the FREQ_LIMIT_RATIOS (MSR 1ADh) bits [7:0]. For unlocked processors, the IA FW has visibility and also the ability to change some of these parameters.

7.2.2 IA FW Responsibilities

Listed below are the responsibilites of IA FW configurations and impacts for Burst Mode:

- Burst Mode Capability Detection
- Hiding Burst Mode Capability
- Enabling Burst Mode
- Setting package power limits and time constants
- Engaging Burst Mode

- Verifying the IACores' Ratio in Burst Mode
- Including ACPI changes to Enhanced Intel® Speedstep Technology-related ACPI structures

7.2.3 Burst Mode Dependencies

Burst Mode performance is constrained by the following factors:

- The number of cores in C6 state - There are factory configured values for the maximum possible Burst Mode upside for N active cores, e.g., 4 frequency steps of upside with only 1 core active, only 2 frequency steps of upside with 2 cores active(For the purposes of Burst Mode configuration, an "active" core is any core which is NOT in C6.)
- The estimated power consumption of the part.
- The temperature of the part.
- The ICCmax of the Voltage Regulator.

Any of these can set the upper limit of Burst Mode frequency for a given workload. These limits apply like TM2 mechanism used in Adaptive Intel Thermal Monitor. For example, if the power/current/thermal limits are reached, ratio is decremented by 1 on successive short regular intervals until the Max Non-Burst Ratio is reached or until the limits are below the specific limits.

7.2.4 Detecting Burst Mode Support

Burst Mode Capability is detected using a feature flag exposed by the CPUID instruction's Power Management leaf (function 6). However, Burst Mode Capability can be hidden using package-level Burst Mode Disable located in IA32_MISC_ENABLES (MSR 1A0h)

- IA FW Control

The power-on default value of IA32_MISC_ENABLES[38] indicates to IA FW if the Burst features are present. A default value of 1 indicates that Burst features are present and disabled. IA FW will clear the IA32_MISC_ENABLES[38] to 0 in order to enable Burst.

Table 7-4. Module-Level Burst Mode Disable

Action	Register	Bit Field	Description
Read/ Write	IA32_MISC_ENABLE: MSR 1A0h	Bit [38]	**Burst Mode Disable (R/W)** Disables Burst Mode. On Reset, Burst Mode Disable is set to 0 if the Burst Mode feature is unavailable, and is set to 1 if Burst Mode feature is available. This is done in order to guarantee that the processor does not go into Burst Mode until IA FW has had an opportunity to enable this feature. After reset, Burst Mode Disable is Read Only if the Burst Mode feature is unavailable, and read/write is the Burst Mode feature is available. Burst Mode Disable is also used to hide the Burst Mode feature in the CPUID.(EAX=6) feature flag. If the Burst Mode feature is available, the Burst Mode feature flag is returned when the CPUID.(EAX=6):EAX[1] is executed.

- CPUID Enumeration

OS and applications must use CPUID.06H:EAX[1] to detect whether the IA FW has enabled Burst features. If IA32_MISC_ENABLES[38] is set, CPUID.06H:EAX[1] must return 0.

Note: As IA32_MISC_ENABLES[38] is defined per-package, CPUID has to read from the uncore to get MISC_ENABLES[38]. It is the responsibility of the OS or IA FW to manage read-modify-write conflicts cross-core. Setting IA32_MISC_ENABLES[38] on any core will cause Punit to disable Burst operation for ALL cores.

Table 7-5. CPUID Function 6 -- Power Management Parameters

Input Value		Return Value		
EAX	ECX	Register	Description	Value
06h	NA	EAX	Bit [1] Burst Mode Capability	1 = Available 0 = Not available

Table 7-6. Determining Availability and State of Burst Mode

CPUID.(EAX=6):EAX[1] Burst Mode Feature flag	IA32_MISC_ENABLE MSR1A0h Bit [38]	Comments on Burst Mode
0	0	Not available
0	1	Available but hidden
1	0	Available and visible

NOTE: If the Burst Mode feature is available, CPUID.(EAX=6):EAX[1] Burst Mode feature flag is always the complement of the package-level MSR 1A0h Burst Mode Disable.

Example 7-6. Algorithm to Detect Availability of Burst Mode

```
If ((CPUID.(EAX=6):EAX[1] == 0) and (IA32_MISC_ENABLE MSR 1A0h bit [38] ==
0))
{
  Burst Mode is not available in this physical processor package.
      IA FW should not attempt to enable Burst Mode via IA32_MISC_ENABLE
MSR.
  IA FW should show Burst Mode as Disabled and Not Configurable.
}
Else If (CPUID.(EAX=6):EAX[1] == 0)
{
  Burst Mode is available but globally disabled for the all logical
      processors in this processor package.
  IA FW can enable Burst Mode by IA32_MISC_ENABLE MSR 1A0h bit [38] = 0.
}
Else
{
  Burst Mode is available and enabled for all logical processors in this
      Processor package.
}
```

- OS/Driver Control

Certain software workloads may not be able to tolerate the non-deterministic aspects of Burst operation. Software can temporarily disable Burst operation by setting CLOCK_CR_GEYSIII_CONTROL[32] MSR bit. Disabling Burst on any core will cause Burst to be disabled for ALL cores.

Configuration of certain Burst power budget settings is accessible by the OS/Driver software via IOSFSB registers. PKG_POWER_LIMIT and PP0_BURST_POWER_LIMIT configuration registers are duplicated in IOSFSB space for this purpose. The MSR copies of the registers are initialized by IA FW to typical recommended settings and can be overridden with more conservative values by the OS/Driver by programming the corresponding IOSFSB registers. Punit is responsible for reading both copies of the registers and applying the more restrictive settings to the Burst algorithms.

7.2.5 Enabling Burst Mode Support

Burst Mode capability is always engaged at a package level and thus impacts all active cores. In addition, package-level and thread-level Burst Mode Disables exist. However, in both cases, Burst Mode Disables impact all logical processors in the entire package. The package-level Burst Mode Disable is shared by all logical processors and exists in the IA32_MISC_ENABLE (MSR 1A0h). Since it always defaults to Burst Mode Disabled, this allows IA FW the opportunity to verify platform capability and if needed to hide the Burst Mode capability from the OS.

The thread-specific Burst Mode Disable is not intended for IA FW use since it does not impact the Burst Mode feature flag. Instead, the thread-specific Burst Mode Disable is intended for future OS use.

Table 7-7. Thread-level Burst Mode Disable

Action	Register	Bit Field	Description
Read/Write	IA32_PERF_ CTRL :MSR 199h	Bit [32]	**Burst Mode Disable (R/W)** Disables Burst Mode. On Reset, this thread-specific Burst Mode Disable defaults to 0 indicating Burst Mode is not disabled by this logical processor. **NOTE:** When any active (not in Deep Power Down Technology/C6) thread has PERF_CTL[32]=1, then Burst Mode is disabled for the entire package.

For Burst Mode to work on the IA Cores, the IA FW must also enable C-states in the processor.

7.2.6 Engaging Burst Mode

Burst Mode capability is engaged when Burst Mode is available, enabled at module level using IA32_MISC_ENABLE MSR bit [38], enabled at the thread level using IA32_PERF_CTRL (MSR 199h) bit [32], and any active core has requested P0 Burst Mode P-state.

The P0 Burst Mode P-State encoding is the Max Burst Ratio as determined from FREQ_LIMIT_RATIOS MSR bits [7:0]. Based on this new requested P-state, the processor internally calculates the highest performance ratio from all of the requested ratios in a package. Then calculates the correct voltage (VID) for the package to go to, and changes the ratio/voltage appropriately. And based on the current processor conditions, the processor's ratio may dynamically exceed the Max Non-Burst Ratio.

APL SoC do not support each core running at different ratios when Burst Mode engages, thus all active cores will run at the resolved P0 Burst Mode P-state ratio based on the state of the Enhanced Intel® Speedstep Technology Hardware Coordination Disable (refer EDS for more details).

Unless disabled by an active core (where active is defined as a core not in C3, Deep Power Down Technology or C6 state), Burst Mode is engaged when any active core has requested P0 Burst Mode P-state.

7.2.7 Verifying Processor's Ratio while in Burst Mode

All cores in the part will be at the same ratio. Even at opportunistic Burst ratios, all cores will run at the same frequency/voltage. IA32_PERF_STS MSR will reflect the ratio at which the processor is currently operating.

Table 7-8. IA32_PERF_STS MSR 198h While in Burst Mode

Action	Register	Bit Field	Description
Read	IA32_PERF_STS MSR 198h	Bits [15:8]	Current (target) frequency. The clock unit updates these bits. Updated only after transition completion.

To get true Burst Mode performance information, the IA32_MPERF MSR (Refer EDS) and IA32_APERF MSR (refer EDS) should be used. Use CPUID.(EAX=6) to check the Hardware Coordination Feedback Capability feature flag and verify these MSR exist. Refer the below table.

Table 7-9. CPUID Function 6 -- Detecting ACNT/MCNT MSRs

Input Value		Return Value		
EAX	ECX	Register	Description	Value
06h	NA	ECX	Bit [0] Hardware Coordination Feedback Capability (Presence of ACNT, MCNT MSRs)	1 = Available 0 = Not available

The IA32_MPERF MSR and IA32_APERF MSR provide a way for software to determine the actual ratio. For APL SoC, C0_MCNT and the Time Stamp Counter both run at the Max Non-Burst frequency. And C0_ACNT runs at the actual resolved frequency. C0_MCNT can be used to verify the Max Non-Burst Ratio. C0_ACNT can be used to verify the current ratio (including current Burst ratio).

- Execute the CPUID. (EAX=6) to verify the IA32_MPERF MSR (which contains C0_MCNT and IA32_APERF (MSR E7h) (which contains C0_ACNT) are available.
- Clear the C0_MCNT and C0_ACNT registers at the beginning of the reference period.
- Read the C0_MCNT and C0_ACNT registers at the end of the reference period.
- Compute the actual frequency by dividing the C0_ACNT delta by the reference period.

Actual frequency = (Ending C0_ACNT value)/ (Reference period)

7.2.8 Platform Considerations for Burst Mode

APL SoC allows a platform IA FW to configure power limits for the IA Cores (Primary Power Plane) in addition to the package level power limits described in previous section. An IA FW may choose to provide additional setup options to allow end user to configure and program these limits.

The programming steps for the primary plane power limit is identical to the package Burst power limits as described in the previous section.

It is recommended that the IA FW lock various power limit MSRs after programming during boot and resume from Sx states to prevent overwriting during run time. Also on a boot, IA FW needs to program these values before setting the IA FW_RESET_CPL bit.

7.2.9 Burst Thermal Control

For optimized thermal control of both IA and GT components in Burst Mode, IA FW can configure Thermal Control Circuit (TCC) to kick in earlier than the original factory configured PROCHOT# activation temperature. This is achieved by programming the TCC Activation Offset field in the TEMPERATURE_TARGET MSR (MSR 1A2, refer EDS).

Note: This is preferred over using ACPI _PSV reporting as passive throttling states disengage Burst Mode.

7.2.10 Voltage Regulator Current Delivery Considerations

For the processor to operate in full frequency range, system VR must meet Iccmax requirements for the SKU.

7.3 Energy Efficiency Policy and Power Aware Interrupt Routing

Energy Efficiency policy on APL SoC enables a new feature Power Aware interrupt routing (PAIR). The PAIR scheme allows OS/IA FW to choose its preference towards performance improvement or towards power saving when redirecting MSIs. In other words, OS/IA FW can inform its choice such that an MSI can be redirected to an active core if power-saving is preferred or to a sleeping core (after waking up the core) if performance-improvement is preferred. This feature helps save power or performance in interrupt servicing depending on Energy Policy chosen. For example, in an aggressive Burst scenario, the core in non-Burst Mode services the interrupt. Likewise in a low power scenario, cores in deeper C states are not woken up and interrupts are serviced by cores in either C0 or C1 state. Software configurations

- Energy Policy configuration. IA FW/Operating System software need to configure the desired energy policy in IA32_ENERGY_PERFORMANCE_BIAS (MSR 1B0h). This will determine which of the cores will service the interrupt.

7.4 IA FW PCODE Mailbox Interface

APL processors support the mailbox interface for communication between the IA FW and the PCODE. This communication can be used for different Power management tuning as listed.

BIOS MAILBOX INTERFACE **MCHBAR MMIO Address : 0x7084**			
Bit	**Scope**	**Default**	**Description**
31	Package	0	RUN_BUSY (R/W) IA FW must set this bit along with the rest of the command and continue to poll it until it is cleared. This bit is cleared by Pcode once the command is completed

30:29	-	-	Reserved
28:8	Package	0	ADDR (R/W) This field contains the address specific to the mailbox command being used
7:0	Package	0	Command (R/W) This field contains the SW request command

BIOS MAILBOX DATA
MCHBAR MMIO Address : 0x7084

Bit	Scope	Default	Description
31:0	Package	0	Data (R/W) This field contains the data that is required for the specific command

Apollo Lake Mailbox Return Codes

On processing a specific mailbox command, the PCODE returns the status of the command in the bits [7:0] of Mailbox interface register. Below is a list of possible codes that the PCODE may return.

IA FW MAILBOX INTERFACE
MCHBAR MMIO Address :

Bit	Scope	Default	Description
7:0	Package	0	Command completion status (RO)

7.5 Running Average Power Limits

The RAPL algorithm works in conjunction with other algorithms and microarchitectural blocks to improve the overall power management of the processor package. APL being a monolithic CPU package, the RAPL algorithm focuses on the following four domains:

- Package
- IA
- GT
- DRAM

In general the RAPL algorithm focuses on limiting performance based on defined power limits and also a defined time window which is used to assess the exponential weighted moving average of the power limits over a period of time. Each of the above mentioned domains have a separate set of power limits and register interfaces to read the predefined limits, configure the new limits based on predefined limits and verify the energy levels and RAPL algorithm performance. The below table gives an overview of available power limits and interfaces in each domain.

Domain	Power Limits	Energy status
Package	PL1 (MSR/MMIO) PL2(MSR/MMIO) PL3 (MSR/MMIO) PL4 (MSR/MMIO)	Yes
IA	-	Yes
GT	-	Yes
DRAM	-	Yes

7.5.1 Package Domain

7.5.1.1 Register Interface for Package Domain

Predefined limits for package domain are defined through the maximum and minimum power limits in the MSR PACKAGE_POWER_SKU (614h). This MSR also indicates the predefined time window limits for package domain.

The unit of measurements is defined in the MSR PACKAGE_POWER_SKU_UNIT (606h).

Overall Energy consumption in Package domain can be read from the MSR PACKAGE_ENERGY_STATUS (611h).

PKG PL3 and PL4 limits are configured using MSR PL3_CONTROL (615h). IA FW must configure PL3 power limit, duty cycle and time window in bits [14:0], [30:24] and [23:17] respectively while PL4 must be configured in bits [46:32]. Both power limits must be explicitly enabled using bits 15 and 47.

7.5.1.2 Power Limit Configuration in Package Domain

The below pseudocode and flow graph give necessary details on configuration of Package Power limits. Some considerations to keep in mind are

The Package Power Limit 1 is enabled by default in the MSR. IA FW must enable Power Limit 2.

- Failure to specify Power Limit override values will result in default factory configured values in the MSR.
- The Power Limit override value and enable bit can be configured in one single register write.
- The PL1 value configured is usually rounded to closest multiple of 4 by the processor
- All values configured that are not within the factory configured boundaries, are automatically clamped to the hardware limited value.
- The updated Power Limit value will take effect immediately on the next P-state request.

- The clamp bit defined for PL1 and PL2 (bits 16 and 48 respectively) may be used by IA FW to allow processor to lower the performance state to below P1 even when OS is requesting for a Turbo state. It is however recommended that this bit not be set on APL systems in order to operate within spec.

7.5.2 IA Domain

7.5.2.1 Register Interface for IA Domain

Predefined Limits for IA domain are defined through the maximum and minimum Power Limits in the MSR PACKAGE_POWER_SKU (614h). This MSR also indicates the predefined time window limits for IA domain.

The performance and effectiveness of the RAPL algorithm functioning cannot be verified specifically for IA domain. Overall Energy consumption in IA domain can be read from the MSR PP0_ENERGY_STATUS (638h).

The unit of measurements is defined in the MSR PACKAGE_POWER_SKU_UNIT (606h).

7.5.3 GT Domain

7.5.3.1 Register Interface for GT Domain

Predefined Limits for GT domain are not known unlike IA and Package domains. The performance and effectiveness of the RAPL algorithm functioning cannot be verified specifically for GT domain. Overall Energy consumption in GT domain can be read from the MSR PP1_ENERGY_STATUS (641h).

The unit of measurements is defined in the MSR PACKAGE_POWER_SKU_UNIT (606h).

7.5.4 DRAM Domain

7.5.4.1 Register Interface for DRAM Domain

IA FW must configure the Power Limits PL1 and PL2 and time window fields in DDR_RAPL_LIMIT MSR 618h to be used by the processor RAPL algorithm. There are no predefined power limits unlike the Package and IA domain but instead IA FW can configure in DDR_RAPL_CHANNEL_POWER_FLOOR (MCHBAR MMIO 0x5888), the minimum limit below which there should be no DRAM throttling.

The performance and effectiveness of the RAPL algorithm functioning can be verified using the MSR DDR_RAPL_PERF_STATUS (61Bh) which serves as a counter to indicate the accumulated duration during which DRAM throttling occurred due to RAPL. Overall Energy consumption in DRAM domain can be read from the MSR DDR_ENERGY_STATUS (619h).

In addition to the above registers, DRAM domain exposes command weight registers that can be used to configure the energy contribution level of each of the DRAM commands.

The unit of measurements for the power limits is defined in the MSR PACKAGE_POWER_SKU_UNIT (606h) while the unit of measurement for DRAM command weights is configurable in DDR_ENERGY_SCALEFACTOR register MCHBAR MMIO 0x5884.

7.5.4.2 Power Limit Configuration in DRAM Domain

The PL1 and PL2 power limit and time window configurations for DRAM domain are similar to that of Package domain. Both PL1 and PL2 for DRAM domain are disabled by default and IA FW must enable it. In addition to these configurations, IA FW must set the and also configure the weights to define the power budget distribution for each DRAM command. IA FW must configure the Power Floor for both DDR channels at appropriate values such that it does not result in too poor performance or system hang. All these three configurations will be used by the DRAM RAPL algorithm to decide the DRAM throttling during different commands.

Note: Due to dependency on DIMM characterization data, it is recommended that the DRAM command weights be configured in the Memory Reference code and power limit configurations be done post RESET_CPL bit setting.

7.5.4.3 Calculating Value for Time Window

To arrive at the appropriate Time Window value to be programmed in the MSR based on user defined time window in seconds, IA FW uses the below equation.

Time Window = (float)((1+X/4) * (2^Y))

where X corresponds to bits 23:22 of the time window field

where Y corresponds to bits 21:17 in the time window field

The unit of measurement of this Time Window is available from MSR PACKAGE_POWER_SKU_UNIT (606h). IA FW must divide the user entered value by the correct measurement unit and place it in the equation to decipher the values of X and Y.

Time in seconds = (float)((1+X/4)*(2^Y)/(2^10)), This assumes that the default unit

of measurement is 976uS == (1S/(2^10))

For a user defined value of 5 seconds, the equation can be satisfied only when X=01b

and Y=12

5= (1.25)*(2^12)/(2^10)

Note: It is recommended that the time window programmed is greater than 100ms.

7.5.4.4 Power Limit Duty Cycling

APL SoC supports duty cycle throttling of GLM cores and GT through PL1. This feature can be enabled by IA FW mailbox command. WRITE_PL1_DUTY_CYCLE_SETTINGS (0x7). Bit [0] is the MAILBOX_DATA with this command controls the enable/disable of this feature while bits [15:8] and [24:16] refer IA and GT duty cycle respectively.

7.6 CPU Power States (C-States)

Idle power management is achieved through C-states. APL supports C0, C1 (C1E) C6, C7, C7S, C8, C9, and C10. The availability of specific processor C-states must be detected by executing CPUID.(EAX=5).

7.6.1 Detecting Core C-State Support

Detection of C-states for a specific APL SoC can be performed by reviewing the Monitor-Mwait extensions supported. IA FW should use CPUID.(EAX=5) Monitor/Mwait Leaf to determine if the processor supports MONITOR/MWAIT extensions for various APL SoC specific C-states and sub C-states.

Note: The definition and detection of C-states enumerated via MONITOR/MWAIT extensions are processor-specific C-states, not ACPI C-states.

First verify enumeration of Monitor-Mwait extensions is supported by executing CPUID.(EAX=5):ECX[0].

Then review the number of Monitor/Mwait extensions for each APL SoC -specific C-state. The steps to detect APL SoC -specific C-states are as follows:

If CPUID.(EAX=5):ECX[0] = 1, then this specific processor supports Enumeration of Monitor/Mwait extensions.

If CPUID.(EAX=5):EDX[7:4] = 1, then C1 is supported.

If CPUID.(EAX=5):EDX[7:4] >= 2, then C1E is supported.

If CPUID.(EAX=5):EDX[11:8] >= 1, then C3 is supported.

If CPUID.(EAX=5):EDX[15:12] = 1, then only short latency C6 is supported.

If CPUID.(EAX=5):EDX[15:12] >= 2, then both long and short latency C6 is supported.

If CPUID.(EAX=5):EDX[19:16] = 4, then both long and short latency C7 & C7s are supported.

If CPUID.(EAX=5):EDX[19:16] = 3, then long and short latency C7, short latency C7s are supported.

If CPUID.(EAX=5):EDX[19:16] = 2, then long and short latency C7 is supported.

If CPUID.(EAX=5):EDX[19:16] = 1, then short latency C7 is supported.

If CPUID.(EAX=5):EDX[23:20] > 0, then C8 is supported.

If CPUID.(EAX=5):EDX[27:24] > 0, then C9 is supported.

If CPUID.(EAX=5):EDX[31:28] > 0, then C10 is supported.

7.6.2 Detecting Package C-State Support

The range of package C-states supported for a specific APL SoC can be determined via the power-on default value of the Package C-State Limit contained in bits [2:0] in PMG_CST_CONFIG_CONTROL (MSR 0E2h).

7.6.3 Enabling C-States

C-state support can be configured by the IA FW via the control bits in the PMG_CST_CONFIG_CONTROL (MSR E2h, refer EDS), the PMG_IO_CAPTURE_BASE (MSR E4h, refer EDS), and the POWER_CTL (MSR 1FCh, refer EDS). By default, C-state support via native processor instructions (i.e., Monitor and Mwait) is enabled, but Enhanced C-state support (e.g., C1E) and IO MWAIT Redirection are both disabled. It is IA FW responsibility to enable Enhanced C-state support and IO Mwait Redirection, if required.

In addition to enabling specific C-state support features, IA FW is required to report capabilities to an ACPI OS that supports processor power states in the ACPI namespace. The sequence of steps needed to enable C-states is outlined as below:

1. Verify the supported Core C-states supported for each processor package. The supported Core C-states are consistent for all logical processors in a specific processor package so this step only needs to be done once per package. Refer Section 7.6.1 for details. If no C-states beyond C1 are available, then C-state initialization is complete.
    a. If the above check in Step 1. For verifying the supported Core C-states indicates that C1E is available, and then enable C1E support in the POWER_CTL. C1E Enable has package scope so this only needs to occur for one logical processor in each package.
    b. If the above check in Step 1. For verifying the supported Core C-states indicates that C3 or greater is available, setup IO MWait Redirection support. Refer Section 7.10 for details. The related MSR have core scope so IO MWait Redirection must be enabled for each core in each package.
2. In order to support an OS that does not attempt to discover, wake and manage all logical processors in all packages, the IA FW must place all logical processors other than the system BSP into a state that will prevent impacting the OSPM usage of C-states.
    a. [Recommended if IA FW has prior knowledge that logical processors will not be discovered/managed by OS] The IA FW should hardware disable all logical processors that will not be discovered, awakened and managed by the OS. Leaving unused logical processors enabled but in a deep sleep state is not optimal from a power or performance standpoint.
    b. [Recommended if IA FW does not have prior knowledge whether logical processors will be discovered/managed by OS] The IA FW must place all processors other than the system BSP into a deep sleep C-state which prevents hardware coordination from impacting OSPM management of C-states. Force all processors other than the System BSP to the deepest supported C-state using a MWAIT loop. Wake each logical processor other than BSP and leave it in MWAIT loop which invokes the deepest supported C-state.

c. [Not recommended] If IA FW leaves any logical processors in Wait for SIPI or CLI-HLT loop instead of a MWAIT loop and the OSPM that does not wake discover, wake and manage these logical processors, then the OS may not be able to use EIST and package-level C-states properly.

3. Build the ACPI namespace objects required to report C-state support to the OSPM. Refer Section 7.6.4 for more details.

7.6.4 Building ACPI Structures

Required ACPI namespace objects for APL SoC with the C-States feature:

- _OSC and _PDC Methods
- _CST Method
- _CSD Object

7.6.4.1 _OSC (Operating System Details) and _PDC (Processor Driver Capabilities)

_OSC and _PDC Methods are both required. The _OSC and _PDC methods are used to confirm the OSPM capabilities including the support for MP C-States. Call the _OSC method from the _PDC method to prevent duplication. Refer the ACPI specification for background details on these ACPI methods. The reference document for the Intel-specific _PDC definitions is the Intel Processor Vendor-Specific ACPI: Interface Specification, which is available online at http://www.intel.com/technology/iapc/acpi/downloads.htm.

The _PDC flags that impact the IA FW configuration of C-State related ACPI structures are listed in table below.

Bit [1]	OSPM supports the C1 “I/O then Halt” FFH sequence for multi-processor configurations.
Bit [3]	OSPM is capable of independent C1, P State and T State support for each processor for multi-processor configurations. This bit does not indicate support for asymmetrical _CST, _PSS, or _TSS objects for individual processors in the system.
Bit [4]	OSPM is capable of independent C2 and C3 state support for each processor for multi-processor configurations. This bit does indicate support for asymmetrical _CST objects for individual processor in the system.
Bit [6]	OSPM is capable of native C State software coordination using IA FW supplied _CSD information for multi-processor configurations.
Bit [8]	OSPM is capable of performing native C State instructions (beyond halt) for the C1 handler in multi-processor configurations
Bit [9]	OSPM is capable of performing native C State instructions for the C2/C3 handlers in multi-processor configurations

Table 7-10. _PDC Bit Definitions for C-States

A flowchart of the flow for interpretation of the _PDC parameters is presented in Figure 7-1. This flowchart describes the IA FW steps which must occur in the _PDC method to determine which ACPI methods and objects should be exposed to the OSPM.

7.6.4.2 _CST (C States)

The ACPI _CST package comprises of the C states to report to Operating System. IA FW must pass the list of C states, corresponding latency and power for each state for the Operating System idle state power management. Recommended C-State Configurations.

7.6.4.3 Tablet Platform Recommended C-State Configurations

This section contains the C-State Configurations recommendation for Tablet platform using APL SoC. The recommendation is compatible with ACPI revision 2.0 or later.

Depending on the operating system's support for MWAIT instructions (FFH support in ACPI C-State), two sets of recommendations is provided for reporting the processor C-states to the Operating System. One set of ACPI C State mappings is recommended to maximize battery life while in battery powered operation. A separate set of recommendations is made for AC powered operation, maximizing power efficient system performance with Intel Burst technology.

Table 7-11. Tablet Platform Recommended C-State Configuration

C State Configuration	Power Mode	OS MWAIT Support	ACPI C1	ACPI C2	ACPI C3
Tablet Platform Recommended	Battery	Yes	MWAIT(C1)	MWAIT(C6L)	MWAIT(C10
		No	HLT	LVL3*	LVL*
	AC	Yes	MWAIT(C1)	MWAIT (C6L)	MWAIT(C10
		No	HLT	LVL3*	LVL4*

C state	Exit Latencies in us
C1	1
C6L	50
C10	150

Note: I/O Redirection of legacy P_LVLx C State triggering is needed.

Note: Exposing MWAIT(C6L) to the Operation System in AC Mode for S0ix state and faster battery charging.

Operating System Power Management (OSPM) supporting ACPI re-evaluates C-State Configuration when there is a power policy change. The recommended ACPI C-State mappings for both AC and Battery are:

If MWAIT is supported is as follows:

Expose MWAIT(C1) as ACPI C1.

Expose MWAIT(C6L) as ACPI C2.

Expose MWAIT(C10) as ACPI C3

If MWAIT is not supported, then I/O redirection of legacy P_LVLx C State triggering must be enabled and supported as follows:

Expose HLT as ACPI C1.

Expose P_LVL3 (mapped to CPU C7) as ACPI C2.

Expose P_LVL4 (mapped to CPU C10) as ACPI C3.

If the system includes a bus master device that cannot tolerate long bus master latency, MWAIT(C7) and MWAIT(C10) should be reported with BM_STS avoidance bit set.

7.6.4.4 C-State Configurations and Legacy Operating Systems

The above Tablet C-State configurations work with contemporary Windows* and Linux* operating systems. Some legacy operating systems, such as old Linux kernel prior to 2.6.21, while supporting ACPI 2.0 C States might have trouble with the recommendation due to local xAPIC clock stopping in CPU C4. IA FW may choose to have a setup option to skip ACPI C2 if support for the selected version of legacy operating systems.

Newer Linux* kernel does not have this problem. Always skipping ACPI C2 is also not recommended, as some legacy Windows* OS (e.g., XP SP2 without QFE) did not support discontinuous ACPI C State mapping.

7.6.4.5 _CSD (C-State Dependency)

This optional object provides C-state control cross logical processor dependency information to OSPM. The _CSD object evaluates to a packaged list of information that correlates with the C-state information returned by the _CST object. Each packaged list entry identifies a dependency domain number for the logical processor's C-states, the coordination type for that C-state, the number of logical processors belonging to the domain and the specific C-state Index. Refer the ACPI Specification Revision 3.0b for full details on _CSD object.

A _CSD object should exist in each processor’s scope. The following _CSD object fields must be determined based on the processor topology and the number of processor packages: Domain, Coordination Type, Number of Processors in the Domain and the specific C-State index.

The Domain is defined as the dependency domain number to which this C-state entry belongs. The domain for all logical processors in a specific processor core must be the same, indicating to the OSPM that all the processors use the same VR and all transition to the same C-state. A system with one physical package will have four _CSD domains. A system with two physical processors will have eight _CSD domains. An MP system with multiple processors will have four domains for each physical package, regardless of how many logical processors in each physical package.

The Coordination Type is defined as the type of coordination that exists (hardware) or is required (software) as a result of the underlying hardware dependency. The coordination type must be 0xFE (HW_ALL) indicating the hardware will perform this coordination.

Number of Processors in this Domain is defined as the number of processors belonging to the domain for this logical processor's C-states. OSPM will not start performing power state transitions to a particular C-state until this number of processors belonging to the same domain have been detected and started.

For APL SoC, all logical processors in the same processor core are in the same domain. So the number of processors in a specific domain equals the number of processors returned by CPUID.(EAX=B, ECX=0):EBX[15:0] for physical processor.

MWAIT Hints

Table 7-12. MWAIT C-State Mappings

Requested C-State	ECX[0]	EAX[7:0]		Requested C-state
	UBNOINT	State [7:4]	State [3:0]	
C0	0/1	1111	xxxx	NOP (skip Mwait)
C1		0000	0000/0001	C1/C1E
C3		0001	xxxx	C3 with no L2 shrink
C6		0010		
			000x	C6 but no L2 shrink
C7	0/1	0011	000x	Subs[3:0]: C7 - C6 incremental L2 shrink
			001x	Subs[3:0]: C7S - C6 with full L2 shrink
C8	0/1	0100	0000	C8 - C7S - S0i1
			0001	Subs[3:0]: C8S – C7S –S0i1 shared L2 caches retained in standby
C9	0/1	0101	xxxx	C7S – S0i2
C10	0/1	0110	Xxxx	C7S – S0i3

C-State Coordination

There are three models of C-state resolution defined by ACPI: SW_ANY, SW_ALL, and HW_ALL. APL SoC only has support for HW_ALL C-state coordination. There is no MSR bit to override this behavior (unlike P-state coordination).

The selection of a specific C-state coordination type (e.g., HW_ALL, SW_ALL or SW_ANY) is how the IA FW communicate the C-state support model to the Operating System Power Management (OSPM) via the ACPI _CSD structures. Refer Section 7.6.4.5 for more details on the _CSD object.

C-State Hardware Coordination

On APL SoC, C-states are writeable at a thread level; however, actions can only be taken on a core basis. The thread C-states are resolved to a core C-state. Then special actions occur if it is the last thread in the core is going to sleep (e.g., MLC flushing, and so on.). All cores resolve to a package C-state which may give additional package-level power savings.

At an architectural level, a logical processor can only make direct transitions to and from the C0 state. It cannot transition directly between any other C-states. For example, a logical processor cannot transition directly from C1 to C3; it must go through C0.

C-State Package Resolution

The package C-state always resolves to the highest power C-state of all of the cores. When all cores are in a low power sleep state, package-level power actions can be taken.

Table 7-13. C-State Package Resolution

If Any Core is In	Then Package Resolved C-State is
C0	C0
C1, and no cores are in C0	C1
C3, and no cores are in C0, C1, or C1E	C3
All cores are in C6/C7 and the L2 has been flushed	C6/C7
All cores are in C7 and SoC are in S0i1	C8
All cores are in C7 and SoC are in S0i2	C9
All cores are in C7 and SoC are in S0i3	C10

7.7 APIC Timer Behavior for Power States

For APL SoC, the Local APIC Timer continues to run in deeper states. This is indicated when CPUID.(EAX=6):EAX[2] is set.

7.8 Platform Clock Enable for OSPM

ACPI 3.0 specification requires that a new bit[15] (USE_PLATFORM_CLOCK) needs to be defined in the FACP table in the IA FW to enable the OSPM to use the platform clock to account for time spent in C4 and lower power C states.

Table 7-14. Bit [15] in FACP Table

Bit [15] USE_PLATFORM_CLOCK	A value of 1 indicates that OSPM should use a platform provided timer to drive any monotonically non-decreasing counters, such as OSPM performance counter services. Which particular platform timer will be used is OSPM specific, however, it is recommended that the timer used is based on the following algorithm: If the HPET is exposed to OSPM, OSPM should use HPET. Otherwise, OSPM will use the ACPI power management timer. A value of one indicates that the platform is known to have a correctly implemented ACPI power management timer. **NOTE:** If a value of zero is present, OSPM may arbitrarily choose to use an internal processor clock or platform timer clock for

	these operations. That is, a zero does not imply that OSPM will necessarily use the internal processor clock to generate a monotonically non-decreasing counter to the system.

7.9 Enabling IO MWAIT Re-direction

IA FW must follow the steps listed below on each logical processor to enable IO MWAIT Redirection:

- Enable IO MWAIT Redirection by setting bit 10 in the PMG_CST_CONFIG_CONTROL (MSR E2h).
- Set the Package C-state Limit for each logical processor via the PMG_CST_CONFIG_CONTROL MSR. The default value for Package C-state Limit is factory-configured, so no change is required unless a smaller package C-state limit is desired.
- Set the range of IO addresses that should be captured and redirected as C-state requests via the PMG_IO_CAPTURE_BASE MSR (MSR E4h).
- Set LVL2 IO base address as the PMBASE + 0x14 via the PMG_IO_CAPTURE_BASE MSR.

7.10 EC-State Demotion / Un-Demotion

APL SoC supports C state Demotion and Un-demotion feature which targets higher power saving. C state demotion refers to the processor entering a shallower C state upon deeper C state request based on the interrupt history. This helps higher power saving by avoiding multiple transitions from deeper to shallower C states. Un-demotion on the other hand is the ability of the processor to switch back to the original requested deeper C state once the interrupt storm is over.

7.11 Function Disable

APL PMC GCR space has two 32-bit registers for function disable that IA FW can configure to disable a specific device. To disable a specific device IA FW must

1. Enable Power Gating, if the function requires configuration to do so.
2. Set the D0i3C MMIO register.
3. Set the D3 register.
4. Disable the PCI function (FunDis) in the PSF via IOSF-SB.
5. Write the appropriate bit in PMC MMIO register FUNC_DISABLE0/1 PBASE+ (0x34).
6. Update the LTRs to reflect infinity.

7.12 ACPI Timer Emulation

Since APL does not have a real ACPI timer, IAFW is required to program MSR 0x121 to emulate ACPI timer.

7.13 8254 Clock Gating

CSM module may use 8254 timer for timer interrupt (INT #8). To support CSM, BIOS should not set 8254 Clock Gating. Customer BIOS could enable/disable 8254 CG through policy “Timer8254ClkGateEn” and recommend to have a setup item to control like below sample

BIOS setup options	AOS/OSX	Windows* 8 or Later	Windows* 7 (w/ CSM)	Linux (e.g. Grub Bootloader or Others)
8254 CG	enabled	Enabled	disabled	disabled

Note that, to get memtest tool running in UEFI shell, BIOS need to disable 8254 clock gating so the 8254 timer can be normal operation.

7.14 BIOS RESET Complete

7.14.1 BIOS_RESET_DONE

IA FW will write the RST_CPL (MCH MMIO 0x7078 bit[0] and read PCODE_INIT_DONE (MCH MMIO 0x7078 bit[8]) which indicates to the CPU Power management function that it has completed to set up all PM relevant configuration and allow CPU Power management function to digest the configuration data and start active PM operation.

The following are required to finish before set RESET_CPL,

- Load patch
- MRC
- Finish all BIOS to Punit configurations needed. This is not required to boot, but may be required based on platform needs
- Set P_CR_THERMAL_DEVICE_IRQ_0_0_0_MCHBAR if needed
- Set PUNIT Interrupt Pin(config bus 0, dev0, fun 1, offset 0x3d)
- Set PUINT IRQ to 24 and INTPIN LOCK (MCH bar offset 0x7004)
- Set up P_CR_TEMPERATURE_TARGET_0_0_0_MCHBAR if needed
- Set up P_CR_CORE_DISABLE_MASK_0_0_0_MCHBAR to disable core if needed
- Finally, set BIT0 in P_CR_BIOS_RESET_CPL_0_0_0_MCHBAR to indicate that reset is complete
  - Set the BIOS_RESET_CPL.RST_CPL (bit 0)
  - Poll BIOS_RESET_CPL.PCODE_INIT_DONE(bit 8)

Setting the bits in BIOS_RESET_CPL allows the P unit firmware to start executing any PM/Reset related flows.

When the BIOS_RESET_CPL bit is set, IA FW/Operating system can perform any of these actions as required

1. Transition to Operating System

2. Performance state transitions
3. Idle state transitions
4. Throttle state transitions
5. Turbo power limit configurations including ConfigTDP configurations
6. Overclocking configurations

Note: To program Turbo Power Limits correctly, IA FW needs to read MSR PACKAGE_POWER_SKU (614h) that is configured by the CPU internally after BIOS_RESET_CPL bit is set. It is hence recommended that IA FW program the Power limits at least 1 ms after programming BIOS_RESET_CPL bit to avoid a race condition.

This chapter defines some of the power management concerns that IA FW should consider.

7.15 Power Button Override

IA FW reconfigures the SoC PMU-PWRBTN PIN as a GPIO and reports. PWR BTN is routed via PMIC and PMIC enables PBTN override [10s is for Windows* Blue].

7.16 Additional Power Management Programming

IA FW requires to program the registers listed below for platform power management.

1. Write the right value to the “CF9h Global Reset” bit (PBASE + 0x48[20]). It is recommended to clear the “CF9h Global Reset” bit prior to loading the OS.
2. Lock the “CF9h Global reset” field by setting CF9LOCK bit (PBASE + 0x48[31]).
3. Set “SLPSX_STR_POL_LOCK” register (PBASE + 0x26[2]) to ‘1b’.
4. Enable all clock gating.

7.17 Generate SMI on SLP_EN Write

The SMI on Sleep Enable bits are used to generate an SMI, instead of enter a low power state, when the SLP_EN bit is set, ABASE + 0x04[13]. The SMI_ON_SLP_EN is located at ABASE + 0x30[4] and the SMI_ON_SLP_EN_STS bit is located at ABASE + 0x34[4]. This allows an SMI to be generated upon setting SLP_EN bit instead of putting the system to a suspend state.

7.18 APL RTC Considerations

APL supports standard RTC with a reduced feature set. The minimum features are to get and set time, get and set a wake alarm, and the "update in progress" bit. RTC is located at I/O ports 70/71. APL SoC RTC is functions only when the SoC is S0/S0ix. When the SoC enters off state the onboard PMIC RTC or onboard RTC is expected to maintain the system time. PMIC RTC basically is advantageous from S0ix/idle power savings perspective.

SoC PMC firmware shadows the contents from PMIC RTC to SoC RTC registers on SoC power sequencing. Whenever the host app/IA FW changes the RTC time, PMC would shadow the contents to PMIC over I2C.

7.19 Wake Events in APL SoC

The following registers contain wake event enable bits to control which events will wake the system when in a low power state:

PM1_STS_EN, ABASE + 0x00

GPE0x_EN, ABASE + 0x30~0x3C

Refer APL SoC EDS for more details. The power button is always enabled as a wake event. If the power button is held for more than 4 seconds, APL SoC clears the PWRBTN_STS bit, sets the PRBTNOR_STS bit, the system transitions to the S5 state, and only PWRBTN# is enabled as a wake event.

In the ACPI FACP table, ACPI GPE0_BLK should be reported as ABASE + 0x20 and GPE0_BLK_LEN as 16.

Note: Most wake events will generate an SCI when SCI_EN is enabled or an SMI if SCI_EN is not enabled, on resume. For some GPI wake events the corresponding bits in the GPIO_ROUT register determines the generation of an SCI or SMI. For wakeup from S5, the IA FW should clear the status bit corresponding to the wake event before enabling the SCI or SMI to prevent the SCI/SMI generation. For wakeup from S3, S4 in an ACPI system, IA FW should not clear the status bit corresponding to the wake event before returning control to the OS.

7.20 Low Power Config

The IA FW must perform the following registers programming guidelines to enable low power features in various interfaces.

7.20.1 GPIO

- IA FW to enable participation in dynamic local clock gating and partition clock gating by setting each community MISCCFG offset 0x10 bit[1:0].

7.20.2 P2SB

- Enable PGCB Clock Gating by setting P2SB PCI register 0xE0 bit[16].
- Set Power Control Enable by P2SB PCI register 0xE4 bit[5][3][0].
- Clear Primary Clock Gating and Request bits by P2SB PCI register 0xE8[1:0].
- Clear Sideband Clock gating and Request bits by P2SB PCI register 0xEA[1:0].

7.20.3 PMC

- Enable PMC device D3Hot and DevIdle by setting PCI register 0xA0 Bit[18:17].

- Enable PWM device D3Hot and DevIdle by setting PCI register 0xA0 Bit[18:17].
- Setting PMCTL register 0x1D0 Bit[5:0] by IOSF-SB port

7.20.4 ITSS

Enable ITSS PMCTL by setting IOSF-SB port 0xD0 offset 0x3300 BITE[4:0].

7.20.5 LPSS

In order for the clocks, clock gating and power gating functions to operate properly, IA FW must program the registers in the following order:

14. Set GPPRVRW2 register 0x0604 bit [11][1]by IOSF-SB port.
15. Set PMCTL register 0x01D0 bit [5:0]by IOSF-SB port.
16. Enable Device PG Config per function by PCI CFG register 0x00A0 [18:16].

7.20.6 xDCI

- Set GUSB2PHYCFG register 0xC200 bit [6][1]by IOSF-SB port.
- Set PMCTL register 0x01D0 bit [5:0]by IOSF-SB port.
- Enable Device PG Config per function by PCI CFG register 0x00A0 [18:17]
- xHCI

The basic low power config settings are included below:

- Set DUAL_ROLE_CFG0 register 0x80D8 bit [21:20]by xHCI MMIO.
- Program XHCC1 register bit[21:19] to 3’b110 by PCI CFG 0x40.
- Set XHCC1 register bit[18] to 3’b110 by PCI CFG 0x40.
- Program XHCC1 register bit[10:8] to 3’b001 by PCI CFG 0x40.
- Program XHCLKGTEN register bit[28:0] to 0x0FDF6D3F by PCI CFG 0x50.
- Set PCE register bit [21] [19:17] by PCI CFG 0xA0.
- Clear PCE register bit [16] by PCI CFG 0xA0.
- Set SSCFG1 register bit [17][14] by PCI CFG 0xA8.
- Clear xHCI Aux Clock Control register bit[19][18][16][13:8][1:0]by MMIO offset 0x816C.
- Set xHCI Aux Clock Control register bit[17][14][5:2]by MMIO offset 0x816C.

7.20.7 USB2 PHY

USB2 PHY registers are accessed via IOSF-SB port. All register should use opcode 0x6 for read, 0x7 for write.

- Set USB2 Global Port register 0x4001 bit [25][13][12] by IOSF-SB port
- Program UTMI misc register 0x4108/0x4209/0x4308 per port by IOSF-SB as below.
  - Bit [10:9] -> 0x1

- Bit [7] -> 0x1

Register	Type	Offset	Field/Bits	Write Data
CMNREG16	IOSF-SB	0xF03C	[0]	1’b1
IMPREG22 Lane 0	IOSF-SB	0x1054	[26:24]	3’b111
IMPREG22 Lane 1	IOSF-SB	0x3054	[26:24]	3’b111
IMPREG22 Lane 2	IOSF-SB	0x5054	[26:24]	3’b111

- Bit [6] -> 0
- Bit [5:4] -> 0

- Program USB2 register 0x4100/0x4200/0x4300 per port by IOSF-SB as below.
  - Set Bit [31:26]
  - Clear Bit [24:23]
  - Clear Bit [21:20]
- Program USB2 Global Port 2 register 0x402C by IOSF-SB as below.
  - Set Bit [27]
  - Clear Bit [22]

7.20.8 SSIC MPHY

SSIC MPHY registers are accessed via IOSF-SB port. All register should use opcode 0x6 for read, 0x7 for write. IA FW should programing related registers as below,

egister	Type	Offset	Field/Bits	Write Data
CMNREG20	IOSF-SB	0xF080	[8]	1’b1

For APLK A stepping,

Register	Type	Offset	Field/Bits	Write Data
CMNREG4	IOSF-SB	0xF00C	[5:0]	6’b100011
CMNREG20	IOSF-SB	0xF080	[1:0]	2’b11
IMPREG8 L0 IMPREG8 L1 IMPREG8 L2	IOSF-SB	0x101C 0x301C 0x501C	[25:24]	2’b11

For other stepping,

Register	Type	Offset	Field/Bits	Write Data
CMNREG16	IOSF-SB	0xF03C	[0]	1'b1
IMPREG22 Lane 0	IOSF-SB	0x1054	[26:24]	3'b111

Register	Type	Offset	Field/Bits	Write Data
CMNREG4	IOSF-SB	0xF00C	[5:0]	6'b100011
CMNREG20	IOSF-SB	0xF080	[1:0]	2'b11
IMPREG8 L0	IOSF-SB	0x101C	[25:24]	2'b11

7.20.9 SCC

The SCC controller includes SD card, eMMC, USF, SDIO functions, IA FW should program related registers **per function**.

- Set Device PG config by PCI CFG register 0xA bit[18:17]
- Set UFS PCI CFG register 0xA bit[21] (UFS only).
- Set PMCTL register 0x1D0 bit[5:0] by IOSF-SB port.
- Set GPPRVRW1 register 0x600 bit[22][19][18][16:0] by IOSF-SB port.

7.20.9.1 UFS MMIO Registers Setting

IA FW should program UFS MMIO register 0x858 bit [11:0] to 0.

7.20.9.2 UFS MPHY Low Power Setting

- Set CMNREG16 register bit 0 via IOSF-SB port offset 0xF03C.
- Set IMPREG22 Lane 0 register bit[26:24] via IOSF-SB port offset 0x1054.
- Set IMPREG22 Lane 1 register bit[26:24] via IOSF-SB port offset 0x3054.
- program CMNREG4 register [5:0] to 6'b100011 via IOSF-SB port offset 0xF00C.
- Set CMNREG20 register bit[1:0] via IOSF-SB port offset 0xF080.
- Set IMPREG8 L0 register bit[25:24] via IOSF-SB port offset 0x101C..
- Set IMPREG8 L1 register bit[25:24] via IOSF-SB port offset 0x301C

7.20.10 Audio

IA FW should set FNCFG register via IOSF-SB port at offset 0x530 bit 5 and bit 3.

7.20.11 SMbus

- IA FW should set SMBus PCI CFG register 0x80 bit 18 and bit 1.

- IA FW should set HAE bit to enable automatically power gating via SMBus IOSF-SB port offset 0x10 bit5.

7.20.12 LPC

- Enable LPC PCE (Power Control Enable) by setting IOSF-SB port 0xD2 offset 0x341D bit3 and bit0.
- Enable LPC CCE (Clock Control Enable) by setting IOSF-SB port 0xD2 offset 0x341C bit [3:0].

To achieve maximum power saving, system BIOS should enable LPC CLKRUN# by set LPC PCI offset E0h[0] = 1b.

However, some old LPC devices might not support it, please contact LPC device vendor to know device capability.

7.20.13 SATA

IA FW is requested to program related registers as below,

- Program Common Test Mode 2 register (SIR offset 0x9C) to 0x20DC0224.
- Program Common Test Mode 3 register (SIR offset 0xA0) to 0x004C8000.
- Program Common Test Mode 4 register (SIR offset 0x9C) to 0x00194060.
- Program Common Test Mode 5 register (SIR offset 0x9C) to 0x000F0020.

7.20.14 ISH

Enable ISH PM Control by setting IOSF-SB port 0x94 offset 0x1D0 BIT[5:0]

7.21 S0ix States

APL SoC support S0ix feature. These power states are marked by the following features:

- All devices are in RTD3.
- The CPU is in C7 state, memory retained with no DMA access.
- Any activity in the NC will have to go through a wake event from the SC.

Since in S0ix states the IA is not available, the IA FW role should be in preparing the grounds for these states and configuring the system's behavior the PMC and P-Unit will implement.

7.21.1 S0ix States on APL SoC

SoC support below S0ix states and ACPI Global System Power States:

State	Definition	Device Requirement
S0	System Active. This is the normal operating state of the processor. In S0, the core processor will transition in and out of the various processor C-States and P-States.	N/A
S0i1	CPU in C8. NC devices(Gen, Display, VED, Camera) and all SC devices (except TXE , LPE and communication subsystems) enter RTD3 Asynchronous wake events from communication devices,Audio voice trigger event can wake up the system	CPU->C8 IGD->RTD3 Display->RTD3 Camera->RTD3
S0i2	All the above + LPE and TXE enter RTD3 Asynchronous wake events from communication devices, push button wake up the system	CPU->C9 IGD->RTD3 Display->RTD3 Camera->RTD3 LPE->RTD3 TXE->RTD3
S0i3	All the above + XOSC is turned off Asynchronous wake events from comm devices, push button wake up the system large resume latency due to Stabilization of XOSC once turn on.	CPU->C10 IGD->RTD3 Display->RTD3 Camera->RTD3 LPE->RTD3 TXE->RTD3 XOSC->off
S4	Lowest Power, longest wake latency sleeping state. OS context in Memory is saved into disk. All devices are powered off. Hibernate mode	Off
S5	“Soft off”. All devices are powered off. System Shutdown	Off

7.21.2 S0ix Entry and Exit

The following points need to be taken care of prior to entering S0ix:

- Make _CST tables for ACPI with the right C-state setup.
  - IA FW should ensure that none of these entries have a wake latency that is intolerable for onboard devices in the system.
  - Windows* will use 3 entries from the _CST (the three deepest C1, C2, C3 entries). Expose C1/C4/C7 in both AC/DC scenarios.
  - OS is then responsible to execute C6FS with S0i3 substate on all CPU threads.
- The mandatory condition for the APL SoC to enter S0ix is setting all devices to RTD3 or disabling state prior to getting into S0.PC7 state. The IA has to make sure that this happens.

Enable S0ix in the registers:

- Set the lowest level possible Max S0i state number (ETR.max_s0ix, PBASE+0x48[17:16].

- Inform OS the platform S0ix capability by setting the flags in FADT: LOW_POWER_S0_IDLE_CAPABLE
- Inform OS the platform profile in FADT: set Preferred_PM_Profile to 8 Tablet.

7.21.3 IA FW Role S0ix

- IA FW has to first inform the OS that underneath platform is S0ix capable by setting the "LOW_POWER_S0_IDLE_CAPABLE" in FADT table.
- IA FW has to inform the Platform profile to OS in this case it shall be a Tablet
- OS must be able to enter a C7 which is a base to trigger the S0ix. Thus IA FW must expose C1/C4/C7 on both AC/DC scenarios.
- IA FW must implement the required RTD3 related entry/exit ACPI methods for all those devices that support RTD3.
  - Based on OS capabilities _OSC IA FW to view the max RTD3 state supported by the host operating system i.e. RTD3H or RTD3C, this info is handy for _PS3.
  - ACPI methods needs to be supported are _PS0 and _PS3.
  - These two methods would perform writes to MMIO BAR1 register of ACPI enumerated devices to trigger D3 entry/exit.
  - Since these are MMIO writes, _PSX methods must perform a MMIO read on these registers to ensure posed writes are committed.
  - _At times _PSX methods to have save/restore some of the device specific configuration space.
- IA FW may have to implement power RAIL management methods to conserve more power by turning on/off the PMIC Power or platform discrete powers RAILS through some FET switch during RTD3.
  - ASL methods involved are _ON / _OFF or _PR0/_PR3.
  - ASL will have to invoke the PMIC opregion services to power manage the rails.
- IA FW also has to express the device wake capability and may have to provide the associated wake handler. ACPI methods involved are _DSW.

§

8 Energy Management

Energy Management is an important aspect of tablet platforms which deals with battery charging and reports various stages of the system. The Energy Management architecture involves managing of energy through various hardware components such as PMICs, fuel gauges and battery chargers. This chapter covers from high-level architecture to charging applet use case and battery management.

8.1 APL High-Level Architecture in Windows*

8.1.1 PMIC Driver

PMIC driver is an OS level driver which contains PMIC read interface. The driver communicates with PMIC charging controller, provides information to control PMIC components via ACPI “OperationRegion” services. This driver performs direct communication with PMIC hardware.

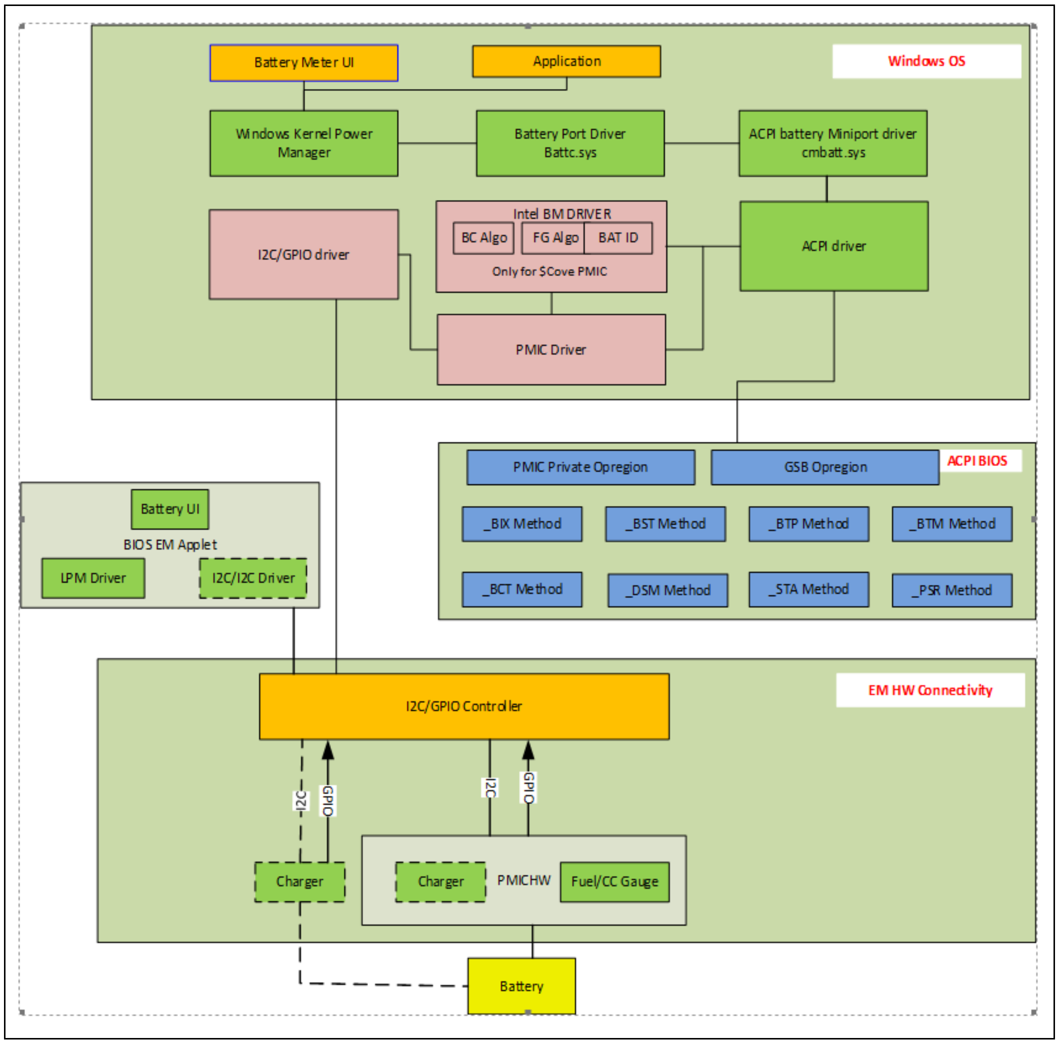
8.1.2 BM Driver

The BM (Battery Management) driver is basically introduced to replace the traditional EC-based microcontrollers. The BM driver performs to update the ACPI operation regions of battery charger and fuel gauge via directly polling devices’ current status. Then the driver also updates with the latest status to ACPI context and exposes to OS application layer. The BM driver only applies for those non-EC or ULPMC based fuel gauges which are integrated into PMIC.

8.1.3 ACPI Support

APL platform supports the ACPI methods and the metholds are used for delivering battery relevant information to battery miniport driver. Take TI and XPOWER PMICS as examples, the ACPI data will be directly updated by BM driver which accesses physical register to get the battery information. However, in non-BM driver case, the ASL methods will read data from PMIC\FG\BC and update to Battery miniport driver. The ACPI 6.0 covers the description of methold in detail.

Figure 8-1. Windows* Energy Management Architecture



8.2 EFI Charging Applet

In order to allow Pre OS charging detection and control, APL platform has added EFI "Charging Applet" as one DXE driver in UEFI. EFI applet is like a normal DXE driver which is launched for switching off the most unnecessary devices and actively updating the charging status on screen. The system is booted with low power mode [LPM] to ensure the system consumes the minimum power and takes the minimum charging time. The EFI charging applet also provides an abstract mechanism to read and write the battery parameters, which will be required to update on UEFI applet screen. The BIOS also provides PMIC libraries to calculate the battery percentage and enable charging during boot time. The Key use cases are:

- Enable Android* charging OS like 'Charging User Interface' under EFI
- Make system boot battery aware and enable EFI to perform charging under low power mode

EFI charging applet is one of UEFI driver which gets invoked upon the charger insertion and displays the battery icon with current capacity. PMIC also wakes up SoC from off state with the charger insertion and let BIOS determine the wakeup reason. If the system wakes coming from the charger insertion, BIOS will activate charging applet. Then charging applet performs battery detection and shows the icon on the screen coming with dimmed system display. This charging applet also polls volume button periodically. If device's volume button is pressed then the applet will get battery information and turn on the screen to show the latest battery information. If power button was pressed, then the applet will reboot the system to OS.

8.2.1 LPM Mode

The BIOS in booting up sequence not only detects the reason of charger sources but also monitors health status of battery pack via the integrated fuel gauge. And base on the battery pack healthy status and charger sources, BIOS will begin to execute charging function. In order to reduce charging time, BIOS would turn off the power hungry devices such as the display panel. Once the user presses the power button, obtain the battery voltage is across the low battery threshold would allow the user to boot the system, otherwise it would continue to charge in the loop and show the current capacity on screen.

To speed up charging and jump to OS, the system also turns off a few power hungry devices which are not required in the S5 charging, so that S5 charging power consumption can be reduced. The devices which are turned off in S5 state are listed as below.

1. IUNIT/GFX 3D.
2. ISH, GMM.
3. I2CS except PMIC/BC and FG/Display I2C.
4. All LPSS SPI/UART controllers.
5. USB XHCI.
6. All PCIe controllers.
7. All SDIO controllers.
8. LPIO1, LPIO2, LPIO spare 1, LPIO spare 2, GMM, HAD, ISH.
9. All CPU cores are set to LFM.

8.2.2 EFI Charging Applet Driver Flow

The EFI charging applet will be launched as a DXE driver and presented in second stage of BIOS. The applet basically makes use of the EFI applet driver protocol services, GOP services, SIMPLETEXT protocol services to enable the charging applet functionality. The system will boot to charging applet in the following two conditions:

- When the system is in S5 state, and charger is plugged in.
- When case system is shutdown with charger connected.

Once the charging applet was triggered and starts charging the battery, the system polls power button at every 100ms. If the volume button is pressed then applet will get battery information and bring up the display to show the updated battery information. If the user presses power button for more than 2 seconds then applet will hand over control to OS loader for system boot.

Overall architecture is shown in the below block diagram

Figure 8-2. S5 Charging Driver Architecture in UEFI BIOS

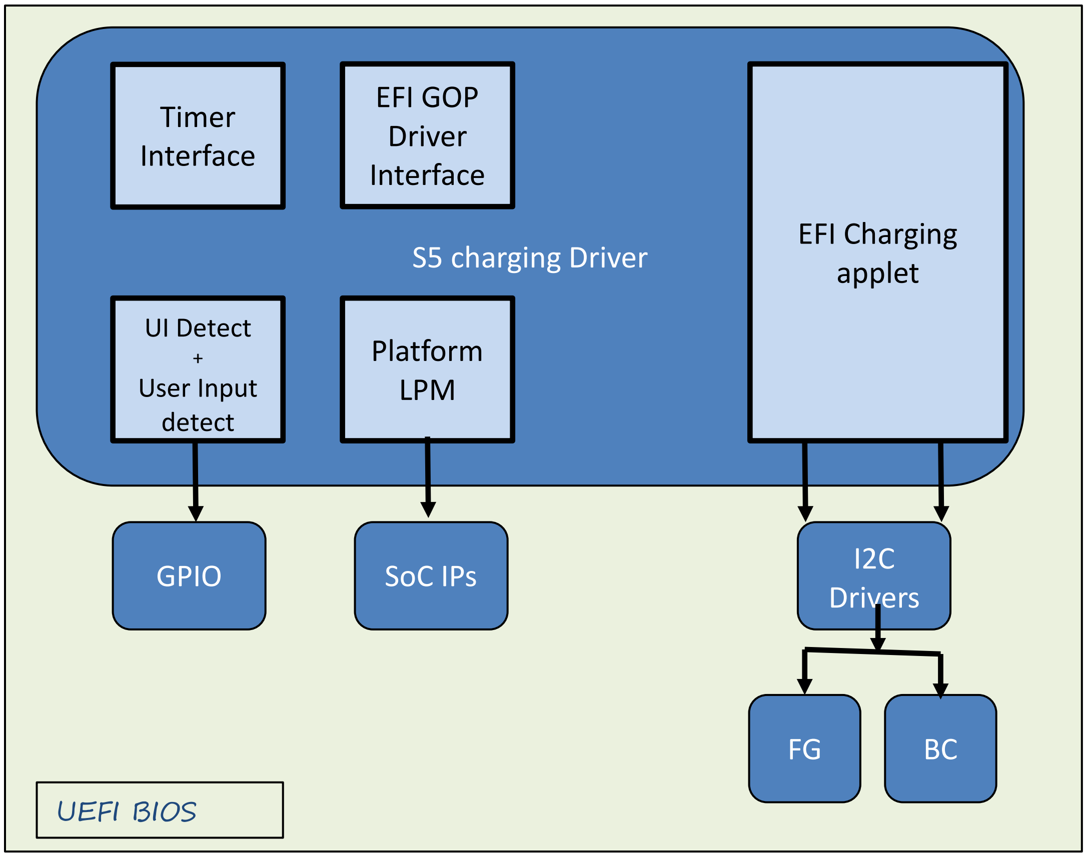
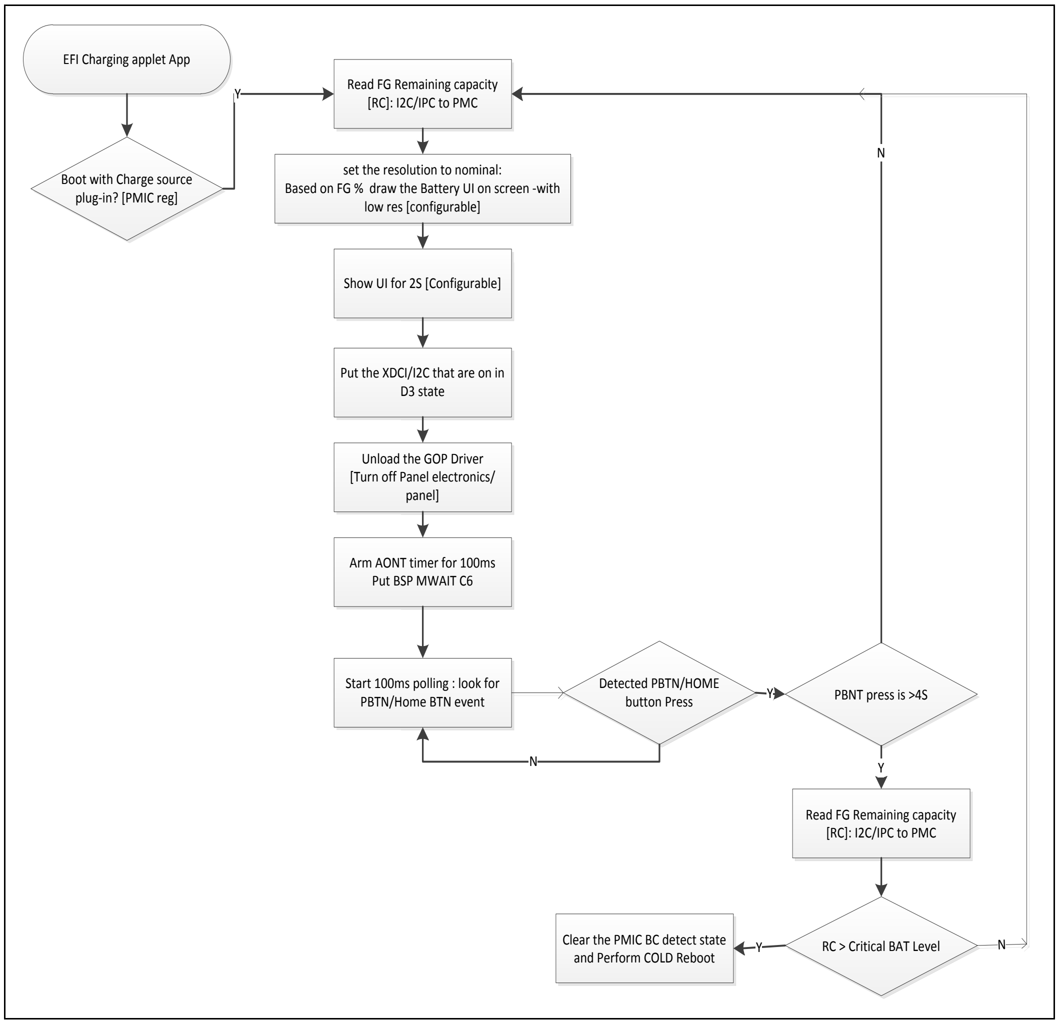


Figure 8-3. S5 Charging Driver Flow


EFI Charging applet App
Boot with Charge source plug-in? [PMIC reg]
Y
Read FG Remaining capacity [RC]: I2C/IPC to PMC
set the resolution to nominal: Based on FG % draw the Battery UI on screen -with low res [configurable]
Show UI for 2S [Configurable]
Put the XDCI/I2C that are on in D3 state
Unload the GOP Driver [Turn off Panel electronics/ panel]
Arm AONT timer for 100ms Put BSP MWAIT C6
Start 100ms polling : look for PBTN/Home BTN event
Detected PBTN/HOME button Press
N
Y
PBNT press is >4S
N
Y
Read FG Remaining capacity [RC]: I2C/IPC to PMC
RC > Critical BAT Level
N
Y
Clear the PMIC BC detect state and Perform COLD Reboot

8.3 EFI Charging Applet Porting

EFI charging applet or the S5 charging is modular and scalable architected. S5 charging code also supports a mechanism to support multiple hardware platforms with the same code. This section covers EFI charging applet function which needs to be ported for implementation of OEM designs.

8.3.1 S5 Charging State Machine

The state machine layer is responsible for managing the charging state and driving the flow of charging irrespective of hardware connected.

8.3.1.1 S5 Charging Entry Point

This is an entry point for the EFI Charging Applet. This verifies the support for S5 charging and invokes the corresponding function to identify the hardware, set charging and create required events to drive the charging.

```
EFI_STATUS
EFIAPI
S5ChargingEntry (
IN EFI_HANDLE        ImageHandle,
IN EFI_SYSTEM_TABLE  *SystemTable
)

Arguments:
ImageHandle             Image handle of the loaded driver
  SystemTable             Pointer to the System Table

Returns:

  EFI_SUCCESS             Thread can be successfully created
  EFI_OUT_OF_RESOURCES    Cannot allocate protocol data structure
  EFI_DEVICE_ERROR        Cannot startup the driver
```

8.3.1.2 Discover Platform Hardware Combination

This represents an abstract function which identifies the PMIC type and based on the PMIC type calls respective PMIC's functions to initialize the hardware interface.

```
EFI_STATUS
EFIAPI
DiscoverHardwareCombination (
  UINT8         BoardId,
  PMIC_TYPE     PmicId
  )
/*++

Routine Description:

  Prior to initialization, discover the hardware combination.

Arguments:
   None
Returns:

  EFI_SUCCESS               Thread can be successfully created
  EFI_DEVICE_ERROR          Cannot startup the driver.

--*/
```

8.3.1.3 Initialize S5 Charging

InitializeS5Charging is responsible for initializing the polling timers and creating the event for charging functionality. It registers several handlers for charging events.

```
EFI_STATUS
EFIAPI
InitializeS5Charging(void)

Arguments:
   None
Returns:

  EFI_SUCCESS               Thread can be successfully created
  EFI_DEVICE_ERROR          Cannot startup the driver
```

8.3.1.4 Set Active State

The EFI charging applet can play a role in one of the four states: initial state, charging state, charged state and power up state. The system will normally start in the initial state. Once the charger is found the system mode will be changed to “charging state” till the system is completely charged.   Then the state will be changed to “charged”. In case the user presses power button for a longer time to boot to OS, the charging state will be changed to Power Up state. For each state we have handlers as explained in below functions.

```
EFI_STATUS
SetActiveState(StateType State)

Returns:

  EFI_SUCCESS              Thread can be successfully created
  EFI_DEVICE_ERROR         Cannot startup the driver.
```

8.3.1.5 S5 Charging State Handler

ChargingTimerHandler will be used for controlling the flow of charging states. It will be invoked periodically as value set in BUTTON_ARRAY_TIMER_INTERVAL. It will launch the corresponding handlers based on the charging state.

```
VOID
EFIAPI
ChargingTimerHandler (
  IN EFI_EVENT    Event,
  IN VOID         *Context
)
```

8.3.1.6 Execute Initial State Actions

Once the charging applet is launched, the default mode will be set to initial mode and **ExecInitialStateActions** will be invoked. This will start charging the batteries, and display battery icon on the screen.

The function will change the state of the applet to charged state if system is fully charged. If the system has woken from pressing the power button, the state will be changed to ‘Power Up’. If the battery percentage is lower than 100% and charger was connected, the applet mode will be changed to charging state.

```
EFI_STATUS
ExecInitialStateActions(void)

Returns:

  EFI_SUCCESS             Thread can be successfully created
  EFI_DEVICE_ERROR        Cannot startup the driver.
```

8.3.1.7 Execute Charging State Actions

This function will be invoked when the EFI charging applet is changed to “charging state”. ExecChargingStateActions will monitor the status of charger, power button and current battery percentage. It will also display current battery icon and percentage on the screen. ExecChargingStateActions can change the charging applet to charged state once the battery gets completely charged or it can change to “power up” if the user presses power button for more than 10 seconds.

```
EFI_STATUS
ExecChargingStateActions(void)


Returns:

  EFI_SUCCESS             Thread can be successfully created
  EFI_DEVICE_ERROR        Cannot startup the driver.
```

8.3.1.8 Execute Charged State Actions:

This function is responsible for executing the tasks required in the ‘charged state’. When the system is completely charged ExecChargedStateActions will display battery fully charged icon on the screen and initiate the system shutdown.

```
EFI_STATUS
ExecChargedStateActions(void)

Returns:

  EFI_SUCCESS             Thread can be successfully created
  EFI_DEVICE_ERROR        Cannot startup the driver.
```

8.3.1.9 Execute Power Up State Actions:

This function is responsible for executing the tasks which should be carried out during 'PowerUp state'. When this is invoked, it will cancel the timer event for notification function ChargingTimerHandler and system will be rebooted to OS. The system will change the mode to Power Up only if the battery is out of critical voltage.

```
EFI_STATUS
ExecPowerUpStateActions(void)

Returns:

  EFI_SUCCESS             Thread can be successfully created
  EFI_DEVICE_ERROR        Cannot startup the driver.
```

8.3.2 EFI Charging Applet Protocol

The charging applet protocol provides an abstract mechanism to use the charger, fuel gauge and PMIC interfaces irrespective of the hardware used. This protocol provides support to enable EFI charging, EFI battery and enable fast charging under EFI.

```
struct _CHARGING_APPLET_PROTOCOL {
  CHARGING_APPLET_GET_CHARGER_TYPE           GetChargerType;
  CHARHING_APPLET_SET_CHARGE_RATE            SetChargeRate;
  CHARGHING_APPLET_GET_CURRENT_CHARGE_STATE  GetCurrentChargeState;
  CHARGING_APPLET_GET_BATTERY_INFO           GetBatteryInfo;
  PUT_PLATFORM_TO_LOW_POWER_MODE             PutPlatformToLoPowerMode;
  UINT16                                         MajorRevision;
  UINT16                                         MinorRevision;
};
```

The protocol uses the PmicInterface, InputButtonInterface, DisplayInterface and ChargerInterface as a mode of communication to PMIC, buttons, display and charger respectively.

8.3.3 PMIC Library Interface

The communication of State Machine with the PMIC is via the PMIC Libraries. EFI Charging Applet maintains a list of PMIC pointers containing the hardware connection information as well as interfacing functions which can talk to PMIC.

```
struct PMIC_object *g_pmic_support_list[] = {
  &WcObj,
  &dm_obj,   //dummy
};
```

8.3.4 Charger Interface

The charger interface is been provided to the Charging applet to abstract the power charger functionalities. The charger interface is represented as below.

```
typedef struct {
  CHECK_BATTERY_PRESENCE        CheckForBatteryPresence;
  CHECK_CHRG_PRESENCE           CheckForChargerPresence;
  START_CHARGING                StartCharging;
  GET_STATE_OF_CHARGE           GetStateOfCharge;  // TODO: add current
state of charge to argument
  SET_CHARGE_RATE               SetChargeRate;
  GET_BATTERY_INFO              GetBatteryInfo;
} ChargerInterface;
```

8.3.5 Battery Fuel Gauge Parameters

There are some parameters related to battery calibration such as Full charge capacity, cycle count, learned cycles. These parameters need to be stored in permanent storage and are required to be maintained by the BM driver. BIOS provides a way to store these parameters in the non-volatile memory in the BIOS DXE phase by DPTF Driver and makes them available for the BM Driver. BM Driver directly reads these parameters from NVRAM area.

The Battery dynamic fuel gauge parameters are listed as below:

```
typedef struct {
  UINT32 signature;
  UINT32 fcc;               //Full charge capacity
  UINT16 cycle_count;       //Number cycle the battery fully charged
  UINT32 cycle_q;
  UINT16 learned_cycle;     //NUmbe of cycle used by the Software fuel gauge
algorithm
  UINT16 c_min_rate;        //Charge minimum rate
  UINT32 rmc;               //Remaining capacity
  UINT32 nac;               //Net available capacity
  UINT32 overcharge_q;
} BATT_CONF;
```

8.4 APL UEFI BIOS Battery Management ACPI Support in Windows*

There are two approaches for reporting battery management data through ACPI to the OSPM. The traditional way uses ASL _Exx or _Lxx handlers to which are interrupts handlers generated by fuel gauge or battery charger or an EC based microcontroller.

These interrupt handlers sends notify command after which OSPM executes _BIX or _BST methods. In case of integrated PMIC design, this approach might become complicated and less portable. The other approach uses a battery management driver to poll data from fuel gauge, provide interrupt handlers. Then the battery management driver updates the polled data to ACPI operation region and invokes ASL notification methods.

8.4.1 Battery Management Driver Device

The Battery Management Driver reads the hardware parameters directly from the hardware and updates them to BM driver ACPI operation region. The BM driver will have an ACPI operation region created in ASL for every PMIC. BM driver invokes PTYP method to get the type of PMIC from NVRAM stored variable. This gives the BM driver identification of the PMIC present on the platform.

The BM driver polls ~~for~~ the battery data from the hardware, if this is found to be changed, it invokes the SBTP method. The SBTP method notifies the OSPM that battery data is needed to be reevaluate by _BST method. There might be more than one BM driver operation region present in the ACPI, BM driver will get the correct PMIC through PTYP and identify the correct operation region. By this way it identifies the PMIC type and communicates with it.

In case of S5 charging the charging will be handled by EFI charging applet alone. BM driver or ACPI will not be involved. For details, battery ASL code for BM device which defines hardware ID, compatible ID and relevant component names, Method operations used by BM driver, and operation region for BM device.

8.4.2 Battery Device

The Battery Device contains the ACPI methods which are used to report the static and the dynamic battery parameters to OSPM. The OSPM uses this method to get the current status of battery. The ACPI battery device has various buffers created which contains the battery static and dynamic parameters. In case of battery management driver is present the data will be updated by BM Driver ACPI operation region to SBTP Methods.

If there is no Battery Management Driver the fuel gauge and battery charger will have their own ACPI operation region defined. The _Exx or _Lxx event handlers will be defined to handle the GPIO interrupts raised by the fuel gauge or battery Charger device. This will notify OSPM of any changes in the battery properties, after which the battery ACPI methods will be invoked as needed. The table below lists the ACPI methods which are used by the Battery device.

Table 8-1. Battery Related ACPI Methods

ASL Methods	Requirement
_BIX	Battery Name, capacity and chemistry details.
_BTP	BTP interrupt
_BST	Battery state, Battery remaining capacity, Battery present rate, Battery present voltage
_STA	Battery is present or not

ASL Methods	Requirement
_BCT	Battery charging time
_PSR	Charger present/ Not present

The below example points out how BM Driver parameters are updated to the _BST method.

```
    Method(_BST,0)
    {
         if(LEqual (\_SB.I2C5.TIDR.AVBL, 1)) {
        Store(\_SB.I2C5.TIDR.CHST, Local0)    //Read charging status
        And(Local0, 0x000F, Local0)
        if (LEqual(local0, 0x02)) {
          Store(0x0002,Local1)
        }

        if (LEqual(local0, 0x01))   {  // if discharging
          Store(0x0001,Local1)
        }

        if (LEqual(local0,0x08))    {
          Store(0x0000, Local1)      // charging done by clearing bit2
and bit1
        }
        Store(Local1,Index(BSTP,0))
        Store(\_SB.I2C5.TIDR.CHGC,Index(BSTP,1))       // chg rate
        Store(\_SB.I2C5.TIDR.RMCP,Index(BSTP,2))       //remaining
capacity
        Store(\_SB.I2C5.TIDR.VOLT,Index(BSTP,3))       //Voltage
         }
         Return(BSTP)
    }
```

8.4.3 Fuel Gauge and Battery Charger Device

If the system follows BM Driver approach, the Fuel Gauge and charger IC will be part of the Battery Management operation region. The BM driver will read the hardware parameters and update to ACPI operation region.

If the system does not follow the BM Driver approach a separate ACPI region has to be defined which includes the I2C and registers in details. Also the _Exx or _Lxx methods need to be defined which can handle the battery charger or fuel gauge interrupt events.

```
OperationRegion( DVBC,GenericSerialBus , 0x00, 0x100)
Field(DVBC, BufferAcc, NoLock, Preserve)    Connection(SMBC), // Use the
Resource Descriptor defined above
    Offset(0x00),
    AccessAs(BufferAcc, AttribByte),
    BC00, 8,
    BC01, 8,
    BC02, 8,
    BC03, 8,
    BC04, 8,
    BC05, 8,
    BC06, 8,
    BC07, 8,
    BC08, 8,
    BC09, 8,
    BC0A, 8,
    BC0B, 8,
    BC0C, 8,
    BC0D, 8,
    BC0E, 8,
    BC0F, 8,
    BC10, 8,
  }
```

```
Name(BUFC, Buffer(3) {00,1,00})
CreateByteField(BUFC, 0x02, DATA)
Method(_E02)    {
      If (CondRefOf(\_SB.I2C1.BATC, Local1)) {
       Store(\_SB.I2C1.BC09, BUFC)
        AND(DATA,0x80,Local0)
        If(LEqual(Local0, 0x80)){
            Store(\_SB.I2C1.BC05, BUFC)
            AND(DATA,0xCF,Local0)
            Store(Local0, DATA)         //close the Watchdog.
            Store(BUFC,\_SB.I2C1.BC05)
        }
           Store(\_SB.I2C1.BATC.INTC(), Local0)
            Notify(\_SB.ADP1, 0x80)
            Notify(\_SB.I2C1.BATC, 0x80)
           Notify(\_SB.I2C1.BATC, 0x81)
   }
```

§

9 Thermal Management

APL SoC supports a range of Thermal management features with different monitor and control techniques. This section outlines the specific configurations that IA FW must do to enable some of these.

9.1 Thermal Monitor

The APL SoC has a digital thermal sensors in each CPU core. These thermal sensors provide ~~a~~ means for periodically monitoring the thermal characteristics of each CPU core and to take corrective action whenever the temperature exceeds operating limits.

The die temperature reading from each sensor is periodically compared against programmable and hard-coded threshold. If any of the thresholds is exceeded, various actions can be triggered such as interrupts, clock-throttling, P state transitions, or I/O pin signaling.

9.1.1 Thermal Interrupts

Thermal threshold configuration for APL can be done at core level or package level. It is recommended that IA FW uses the package level triggers and control for effective thermal management.

9.1.2 Core Level Thermal Interrupts

When one of the thermal thresholds specified in MSR IA32_THERMAL_INTERRUPT (19Bh) is exceeded, it can be programmed to trigger as an interrupt. This is a thread level MSR and interrupt trigger and handling is unique to each thread/core. The software interrupt service routine will have to clear the appropriate status bit in IA32_THERMAL_STATUS (19Ch) in order to clear the interrupt condition.

The MSR CPU_THERM_SENS_CFG (675h) [4] allows core-specific thermal events to be configured to trigger interrupts on all cores.

9.1.3 Package Level Thermal Interrupts

When one of the thermal thresholds specified in MCHBAR register PKG_THERM_INTERRUPT (0x7010) is exceeded, it can be programmed to trigger an interrupt. This is a package level register and interrupt trigger and handling is for entire SoC. The software interrupt service routine will have to clear the appropriate status bit in PKG_THERM_STATUS (0x701C) in order to clear the interrupt condition.

The following table summarizes the thermal thresholds and their supported actions and triggers. Each threshold is described in more detail below.

Table 9-1. Thermal Triggers Summary

Thermal Trigger	Control / Description	Actions
Programmable Threshold 1 (per core)	Threshold specified as an offset below TJMAX at which interrupt is triggered. IA32_THERMAL_INTERRUPT (19Bh)[14:8] enabled using MSR IA32_THERMAL_INTERRUPT (19Bh) [15]	Thermal Interrupt per CPU core on low-high and/or high-low transitions ACPI/software use
Programmable Threshold 2 (per core)	Threshold specified as an offset below TJMAX at which interrupt is triggered. IA32_THERMAL_INTERRUPT (19Bh)[22:16] enabled using MSR IA32_THERMAL_INTERRUPT (19Bh) [23]	Thermal Interrupt per CPU core on low-high and/or high-low transitions ACPI/software use
Programmable Threshold 1 (Package level)	Threshold specified as an offset below TJMAX at which interrupt is triggered. PKG_THERM_INTERRUPT (0x7010)[15:8] enabled using	Thermal Interrupt per Package on low-high and/or high-low transitions ACPI/software use
Programmable Threshold 2 (Package level)	Threshold specified as an offset below TJMAX at which interrupt is triggered. PKG_THERM_INTERRUPT (0x7010)[[23:16] enabled using	Thermal Interrupt per Package on low-high and/or high-low transitions ACPI/software use

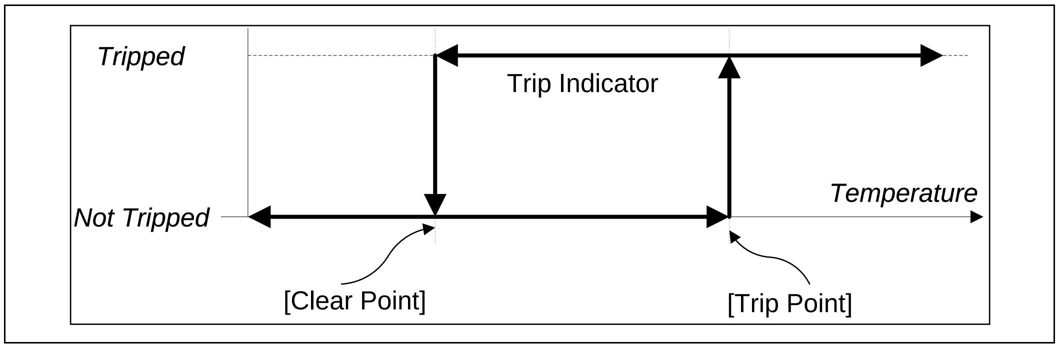
9.2 PROCHOT#

APL SoC supports PROCHOT# as either input only or output only. Bi-directional PROCHOT# is not available on APL.

IA FW may enable PROCHOT# as output pin by clearing MSR POWER_CTL (1FCh) [21]. The HOT condition can be output on the PROCHOT# output pin for system use.

The PROCHOT# input can also trigger a CPU thermal interrupt by setting MSR IA32_THERM_INTERRUPT (19Bh) [2]s. Assertion of the external PROCHOT# input will also trigger TM1 or TM2 throttling mechanisms if they are enabled.

Figure 9-1. Thermal Threshold Hysteresis



9.2.1 THERMTRIP

In the event of a catastrophic thermal failure (such as failure of the cooling system), each thermal sensor can detect that die temperature exceeds thermal specification limits (typically 24-28C above TJMAX) and assert its CAT sensor output. Assertion of CAT must automatically trigger a thermal shut-down of all CPU PLLs, and platform voltage regulators within 500ms and also trigger the SoC THERMTRIP# I/O pin to be asserted.

9.3 MSR Based On Demand Thermal Throttling for Software

Table 9-2. ACPI PROC_CNT I/O Register

I/O Address:		PM_BASE + 10h	
Access:		Read/Write (must be DWORD aligned)	
Bit	Scope	Default	Description
31:5	-	-	Reserved.
4	Package	0	IO Clock Modulation Enable. (R/W) This bit enables modulation of the clock as set in the Duty Cycle field. IO Clock Modulation Enable must be 0 when changing the Duty Cycle field. A value = 0 disables IO emulation based processor clock modulation and a value = 1 enables IO emulation based processor clock modulation. Activating the IO emulation based processor clock modulation immediately modulates the processor clock with the duty cycle indicated by bits [3:1].

I/O Address:		PM_BASE + 10h	
Access:		Read/Write (must be DWORD aligned)	
Bit	Scope	Default	Description
3:0	Package	0	IO Clock Modulation Duty Cycle. (R/W) This field provides the On- demand clock modulation duty cycle. It indicates the clock on to clock interval ratio. Bit 0 is valid only in case of fine grained modulation. Bits[3:0] Duty Cycle 0h Reserved 1h Reserved 2h 12.50% 3h 18.75% 4h 25.00% 5h 31.25% 6h 37.50% 7h 43.75% 8h 50.00% 9h 56.25% Ah 62.50% Bh 68.75% Ch 75.00% Dh 81.25% Eh 87.50% Fh 93.75%

In addition to the Thermal Monitor, clock modulation can be enabled in software by writing to the IA32_CLOCK_MODULATION Model Specific Register which is core-specific. Hardware coordinates the requests from different threads, resolves to the maximum of value (indicating maximum performance). If a thread disables clock modulation throttling, it is not considered in the coordinative process. All threads in a given core are coordinated to minimum frequency. Each processor core will modulate the clock independently.

9.4 Thermal Reporting for Thermal Management

Thermal management in a platform is typically and most importantly done based on the processor temperature. The thermal management solution in a platform can be done either by the platform Embedded Controller or similar external agent, or via ACPI OSPM. There are two kinds of thermal management actions possible, namely, passive thermal management where performance/power of the components are reduced and active thermal management where the system or component fans are activated.

9.4.1 Thermal Reporting for Passive Thermal Management

The temperature value reported/used for passive thermal management must only reflect the respective component's temperature. Since APL SoC is a combination of multiple functions (IA core and Graphics), it is recommended to use only the cores' temperature values for any passive thermal management where IA core P-States are controlled. However, passive throttling cannot handle graphics' temperature. Also since passive throttling disengages burst as soon as it kicks in, it is recommended not to use this technique for burst thermal control.

9.4.2 Thermal Reporting for Active Thermal Management

For active thermal management, the package's max temperature must be used. This ensures the system fan will be activated when any of the functions within the APL SoC exceeds platform specific trip points.

9.4.3 ACPI TSS Throttling States

IA32_CLOCK_MODULATION MSR specifies that APL SoC supports on demand throttling states at a resolution of 12.5% duty cycle. T states can be reported via _TSS to the OS. IA FW needs to report T states from 100% LFM to 12.5% LFM at intervals of 12.5% LFM.

9.5 Memory Thermal Management

The below sample code shows a portion of the implementation in reference code.

```
Name(TSIF                    Package()
                             {
      Package(){100          1000        0         0x00       0}
  Package(){ 88               875        0         0x1E       0}
      Package(){ 75           750        0         0x1C       0}
  Package(){ 63               625        0         0x1A       0}
      Package(){ 50           500        0         0x18       0}
  Package(){ 38               375        0         0x16       0}
      Package(){ 25           250        0         0x14       0}
  Package(){ 13               125        0         0x12       0}
```

§

10 Intel® Enhanced Memory 64 Technology

APL SoC adds support for Intel Enhanced Memory 64 Technology (Intel 64). Intel 64 enables 32-bit software to access a 64-bit linear address space while also allowing compatibility with existing 32-bit software. The Intel 64 adds a new operating mode, with two sub modes - compatibility and 64-bit modes. The features of Intel 64 are:

- Architectural support for 64-bit flat linear addressing.
- 8 new general purpose registers (GPRs).
- 8 new registers for streaming SIMD extensions (SSE).
- 64-bit wide GPRs and instruction pointers.

The Extended Feature Flags function provided when the CPUID instruction is executed with EAX = 80000001h returns the IA-32e extended feature flag in EDX[29]. The Virtual and Physical Address Sizes function provided when the CPUID instruction is executed with EAX = 80000008h returns the actual supported virtual and physical address sizes in a particular IA-32 processor.

10.1 Operating Mode

The processor with Intel 64 can run in either legacy IA-32 mode or IA-32e mode. Legacy IA-32 mode allows the processor to run in protected mode, real address mode or virtual 8086 mode. IA-32e mode is the mode the processor uses when running a 64-bit operating system.

The processor will initially operate in legacy paging-enabled, protected mode and will transition to IA-32e mode once proper bits are set in the IA32_CR_EFER MSR and PAE mode is subsequently enabled. Refer EDS for the definition of IA32_CR_EFER.

Table 10-1. IA-32e Operating Modes

Mode		Operating System Required	Application Recompile Required	Default Address Size (bits)	Default Operand Size (bits)	Register Extension	GPR Width (bits)	Supported by SMM
IA-32e mode	64-bit mode	64 – bit OS	Yes	64	32	Yes	64	Yes (1)
	Compatibility mode		No	32	32	No	32	Yes
				16	16			

NOTE: SMM Supports transitions to/from 64-bit OS and legacy OS. However, PAE and 64-bit linear address are not available inside the SMM environment.

10.2 IA FW Responsibilities

IA FW must detect if the processor is capable of IA-32e and enable it at the appropriate time during POST e.g., during Memory Initialization. IA FW must also ensure that IA-32e is not activated when it hands control of the system over to the OS. In addition, the IA FW must ensure the processor is operating in legacy mode (IA-32e deactivated) before it hands control to an Option ROM during POST.

10.2.1 Activating IA-32e Mode

IA-32e mode is enabled by setting the IA32_CR_EFER.LME bit to 1. IA-32e mode is not activated until software also enables PAE mode and then enables paging.

Software enables PAE by setting the CR4.PAE bit to 1. Physical Address Extensions must be enabled prior to enabling paging (CR0.PG = 1). Switching the processor to IA-32e mode requires several steps and must start in real mode or non-paged protected mode. The sequence to be followed is as listed below:

- Starting from page-enabled protected mode, disable paging by setting CR0.PG = 0. Use MOV CR0 instruction to disable paging (must be located in an identity-mapped page).
- Enable PAE by setting CR4.PAE = 1. Failure to enable PAE will result in #GP fault when an attempt is made to enable IA-32e mode.
- Load CR3 with the physical base address of the Level 4 page map table (PML4).
- Enable IA-32e mode by setting IA32_CR_EFER.LME = 1.
- Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32_CR_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page, until such time that a branch to non-identity mapped pages can be effected.
- Immediately after activating IA-32e mode, the system descriptor table registers (GDTR, LDTR and IDTR) continue to reference legacy descriptor tables. The tables referenced by the descriptors all reside in the lower 4 GB of virtual address space. After activating IA-32e mode, 64-bit operating system software should use the LGDT, LLDT and LIDT instructions to load the system descriptor table registers with reference to the 64-bit versions of the descriptor tables.

10.2.2 Deactivating IA-32e Mode

To return from IA-32e mode to legacy page-protected mode, software must deactivate and disable IA-32e mode using the following sequence:

- Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32_CR_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and the following instructions must be located in an identity-mapped page.
- Load CR3 with the physical base address of the legacy page-table-directory base address.
- Disable IA-32e mode by setting IA32_ECR_FER.LME = 0.
- Enable legacy page-protected mode by setting CR0.PG = 1.

- The instruction following the MOV CR0 that enables paging must be a branch. Both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

§

11 Intel® Virtualization Technology for Directed I/O Initialization

Refer Intel microarchitecture code name Apollo Lake EDS and the Virtualization Technology for Directed I/O Architecture Specification for additional details regarding Intel® Virtualization Technology for Directed I/O (Intel® VT-d). The following sections will address only the IA FW requirements for supporting Intel Virtualization Technology for Directed I/O.

11.1 Verifying Support for Intel® VT-d

Prior to performing Intel VT-d initialization programming, IA FW must verify the platform supports VT-d by reading the CAPID0 register (D0.F0.R0E4h [23]) and confirming it is ‘0b’. If D0.F0.R0E4h [23] is ’0b’, VT-d is supported. Otherwise, VT-d is not supported and IA FW should skip the VT-d initialization steps as well as hiding or disabling any IA FW setup options that allow the user to enable/disable VT-d.

11.2 DMA Remapping Engine Base Addresses

IA FW must assign a base address for each DMA remapping engine in the system agent and enable the decoding of the memory range. These base addresses provide a 4KB window for the remap engine’s register set. There are two DMA remapping engines in the system agent each of which must be assigned a base address and be enabled by setting bit 0 in the following registers:

- DMA Remapping hardware unit #1 (for iGFX) – IA FW must provide a 4KB memory region with the base address programmed in MCHBAR offset 6C88h (for example: 0FED90001h).
- DMA Remapping hardware unit #2 (for DMI/PEG VC0) – IA FW must provide a 4KB memory region with the base address programmed in MCHBAR offset 6C80h (for example: 0FED92001h).
- On APL SoCs, IA FW must perform the below steps in addition to the programming of VTDBAR base addresses.
- After programming DFTVTDBAR, IA FW must read back this register to ensure that the write completes.
- After programming GFXVTDBAR, IA FW must read and write back (the same value) to DEVEN register to ensure that the GFXVTDBAR write completes.

11.2.1 DMA Remapping Hardware Unit Definition (DRHD) Structure

A DMA-remapping hardware unit definition structure uniquely represents a DMA-remapping hardware unit present in the platform. There must exactly be one instance of this structure for each DMA remapping hardware unit supported by the platform. Additionally, there must be at least one DMA-remapping hardware unit definition structure in a platform that supports DMA remapping.

Even if all of the devices associated with a DMA-remapping hardware unit are disabled the System IA FW must include the DMA-remapping hardware unit definition structure for those devices, except for DMA-remapping unit for the internal graphics device. The DRHD structure for the graphics device has to be dynamically added or removed based on the enable/disable state of the device. The DRHD structures for all other devices must be static and present irrespective of the device's enable/disable state.

Refer the Virtualization Technology for Directed I/O Architecture Specification for the latest DMAR Remapping Hardware Unit Definition (DRHD) structure definition.

11.2.2 Reserved Memory Region Reporting (RMRR) Structure

Reserved memory regions indicate system memory regions reserved by IA FW that may be the target of DMA requests while the system software is active. Only the USB and Integrated Graphics devices are expected to utilize the RMRR feature. System software will ensure that the DMA remapping page tables for the respective devices are setup to provide mapping for the specified reserved memory regions with both read and write privileges.

The RMRR regions must be on 4-KB boundaries and the length specified in 4KB increments.

Refer the Virtualization Technology for Directed I/O Architecture Specification for the latest Reserved Memory Region Reporting (RMRR) structure definition.

The platform designer must avoid or limit such regions since these require system software to create a gap in the DMA virtual address range available to system software and its drivers. Hence it is highly discouraged to use the RMRR for reporting other devices than those listed in this section.

11.2.3 Device Scope and DMAR Base Address Reporting

This section describes the platform configuration with the DMA remapping hardware units in the SA.

The IA FW is required to report 2 DRHD structures and the respective base addresses. The Device Scope fields in each DRHD structures are described as below:

- DMA- Remapping Hardware Unit 1 has been lower its scope the integrated graphics controller functions located at B0.D2.F0. This device scope contains two Device Scope Entries identified as: [1, 8, 0, 0, 0, 2, 0], and [1, 8, 0, 0, 0, 2, 1]. The Type field value is 1 to indicate that PCI endpoint devices located at Segment 0, Bus 0, Device 2, Function 0 within the scope of this DMA remapping hardware unit.
- DMA- Remapping Hardware Unit 2 has been lower its scope all the other DMA sources not explicitly lower the scope of the other DMA-remapping hardware units. It also needs to report IOAPIC and HPET device scope entries if interrupt remapping is supported. The DHRD structure for this unit sets the INCLUDE_ALL flag and hence does not contain any PCI Device Scope Entries. If interrupt remapping is supported, it

needs to contain 1 IOAPIC Device Scope Entry identified as [3, 8, 0, 2, 250, 31, 0] and 1 HEPT Device Scope Entries identified as [4, 8, 0, 0, 250, 15, 0].

IA FW must assert that the DMA Remapping Hardware Unit Definition (DRHD) structure include only one device scope structure entry for the entire HPET timer block, and the path (BDF) in the device scope structure must match requester ID of each HPET timer. IA FW should program requester ID of each MSI capable HPET timer identically.

11.2.4 ACPI DMAR Table

This section provides a sample of the ACPI DMAR Table if the format used for reporting is the same as documented in the above sections.

In following example assumes that the USB controllers use a contiguous memory region which requires only one RMRR structure. This is not a requirement for customer platforms, several RMRR structures can be created depending on the memory usage of the USB controllers. There is no requirement that only two RMRR structures be used.

Table 11-1. Organization of DMAR Table

	Description	Comments	Starting Address/Length in Binary Output Table
1	ACPI DMAR Table Header	ACPI Table header with “DMAR” signature	0h (Starting address) 30h (Length)
2	DRHD structure (Intel® VT-d Engine #1)	FED90000h -- Base address (Dev:Func) (2:0) (2:1) – Device Scope	30h (Starting address) 1Ch (Length)
3	DRHD structure (Intel® VT-d Engine #2)	FED91000h – Base address (Dev:Func) (INCLUDE_ALL=1) – Device Scope (ID:Bus:Dev:Func)(2:250:31:0) – IOAPIC Device Scope (Bus:Dev:Func)(250:15:0) (250:15:1) (250:15:2) (250:15:3) (250:15:4) (250:15:5) (250:15:6) (250:15:7) – HPET Device Scope	4Ch (Starting address) 48h (Length)
5	RMRR structure (Compatible & EHCI USB HC’s)	3E2E0000h (Start) 3E2FFFFF (End) (Dev:Func) (29:0) (29:1) (29:2) (29:7) (26:0) (26:1) (26:2) (26:7)	94h (Starting address) 42h (Length)
6	RMRR structure (Integrated Graphics)	3E800000h(Start) 3EFFFFFF(End) (Dev:Func) (2:0)	D6h (Starting address) 24h (Length)

11.2.5 Default Recommendation for Apollo Lake Platform

Intel recommends that OEMs enable Intel® Virtualization Technology by default for Apollo Lake platforms that have the following conditions.

- The platform ships configured to launch a Virtual Machine Monitor (VMM), (such as Citrix XenClient Type 1 Hypervisor).
- The platform ships configured with security software that requires VT-x (such as McAfee® Deep Defender™ using DeepSAFE™ Technology).
- The user/owner explicitly requests Intel Virtualization Technology be set "On" based upon their intention to launch a VMM.
- Intel TXT is enabled on the platform.
  - To enable Intel® Virtualization Technology when TXT is enabled, set both MSR IA32_FEATURE_CONTROL (3Ah)[2:1] = '11b' to enable VMX.
  - To enable Intel® Virtualization Technology when TXT is disabled, set IA32_FEATURE_CONTROL_MSR [2] = '1b' to enable VMX outside of secure environment.

Otherwise, IA FW should ship with Intel® VT and Intel TXT turned off by default.

IA FW Configurations for Intel VT and Intel TXT Enabling

	Intel TXT Enabled	Intel VT-x Enabled	Legacy (Intel TXT and VT-x Disabled)
IA FW Setup Option for Intel TXT	Enable	Disable	Disable
IA FW Setup Option for Intel VT	Enable1	Enable	Disable
SENTER Global Enable MSR 3Ah [15]	12	0	0
SENTER Parameter Control MSR 3Ah [14:8]	0x7F3	0x00	0x00
Enable VMX outside SMX operation MSR 3Ah [2]	1	1	0
Enable VMX inside SMX operation MSR 3Ah [1]4	1	X	0
Lock BitMSR 3Ah [0]5	1	1	1

[1]Intel TXT requires Intel VT to be enabled. The IA FW setup option for enabling Intel TXT should be hidden or unavailable from user if Intel VT is not capable or disabled.
[2]IA FW can set this bit only when Intel TXT is capable on the processor and selected to be enabled by user in the IA FW setup menu.
[3]IA FW may select other values to enable/disable specific SMX instructions, refer Intel TXT IA FW collateral for programing details.
[4]IA FW can set this bit to 1 only when the processor is Intel® TXT capable.
[5]IA FW must set the lock bit before boot to OS.

§

Appendix A: IA FW Workaround

Issue	A usage model in which there is interdependency between IPU and VT-d is not supported.
Stepping	APL Ax/Bx
Workaround	In order to ensure proper platform functionality, the following BIOS configurations can be used: 1. If IPU is detected to be 'disabled' then VT-d can be 'enabled'. 2. If IPU is detected to be 'enabled' then VT-d needs to be 'disabled'.
Impact of Workaround	
Notes	To fix until Cx stepping

Issue	When SD+SDIO function disabled, if eMMC in D3 function disabled, there is no PG happen
Stepping	All stepping
Workaround	BIOS should make sure SD card and SDIO disabled before eMMC is disabled or before eMMC is put into D3
Impact of Workaround	
Notes	SCC

Issue	The PCI configuration register access (BME and PMCSR) are not propagated to the underlying IP ,which impacts the D0/D3 transition and BM transaction
Stepping	APL Ax
Workaround	BIOS should use _PS3 ASL method to issue PMCSR read to PCI function after putting it in D3
Impact of Workaround	
Notes	LPSS, SCS, PWM, ISH, xDCI

Issue	Mismatches setting for SD card pins
Stepping	APL
Workaround	BIOS should set 20k PU on all SD-Card pins except clock
Impact of Workaround	
Notes	SD Card

Issue	Mismatches in the way the STA signals are hooked up inside the GPIO.
Stepping	APL Ax
Workaround	BIOS should set program DEVSLP GPIO pins (gpio_24, gpio_25) so that the default termination resistor is 20K PU.
Impact of Workaround	
Notes	SD Card

Issue	While entering D0i3, ux_ccu_xtal_frm_refclk_req is asserting and de-asserting without clkack handshake.
Stepping	APL Ax.
Workaround	BIOS should set 10us between the isolation and reset from PGCB. The delay between isolation enable and reset assertion is configurable. Default value is 2 pgcb_clk(80ns). BIOS should set to 256 (10us).
Impact of Workaround	Not Visible to end user. only violating clkreq/ack handshake.
Notes	Clock.

Issue	XHCI controller stuck in reset during warm reboot cycles.
Stepping	APL Ax
Workaround	BIOS should program USB Host Controller Misc Reg 2 bit2 =1, set Bit5 for A0 stepping and clear for all other steppings.
Impact of Workaround	
Notes	USB

Issue	CDC (missing double syncs) on side band signals was found while fixing the first set of violations.
Stepping	APL Ax
Workaround	If S0ix disabled, IA FW should clear Bits 4 and 5 of PMCTL register.
Impact of Workaround	not predictable
Notes	S0ix

Issue	Current serge when SRAM enter/exit retention and might cause damage to unit.
Stepping	APL Ax
Workaround	IA FW should set PTDC register bit 30/31.
Impact of Workaround	
Notes	HAD

Issue	Intermittent loss in Audio during Paus/resume operation.
Stepping	APL Ax
Workaround	IA FW should set fifordy_sel to 11.
Impact of Workaround	
Notes	HAD

Issue	EMMC present register card present bit is not set after SCC resuming from power gating.
Stepping	APL Ax
Workaround	IA FW should implement the WA in ASL code: to disable PG, clear CG for ~2ms and restore CG in _PS0, and to enable PG in _PS3.
Impact of Workaround	eMMC detect function
Notes	eMMC

Issue	nc_xosc_dcg_visa[7] signal goes high after entering CS waking the XTAL from remaining shutdown.
Stepping	APL Ax
Workaround	BIOS to disable HDA clock gating on wake clock by setting D14F0.x48[18] = 0
Impact of Workaround	
Notes	HDA

Issue	SMBus does not generate interrupt when PGCB DCG is enabled.
Stepping	APL Ax
Workaround	BIOS to disable SMBus PGCB dynamic clock gating by writing SMBus PCI Cfg offset 80h bit 18 = 1.
Impact of Workaround	
Notes	SMBus

Issue	Audio vnn_req does not deassert after Sx/S3 entry.
Stepping	APL Bx
Workaround	IA FW should program IPGIW register to 000.
Impact of Workaround	The vnn_req holdoff timer does not have sufficient time to count to zero.
Notes	Audio

Issue	eMMC devices may not work when enter/exist D3.
Stepping	APL Ax
Workaround	IA FW should implement the WA in ASL code: to clear SCS bit 6 and 0 in _PS0 with 1ms delay, and to set SCS bit 6 and 0 in _PS3.
Impact of Workaround	eMMC devices function during PG.
Notes	eMMC

Issue	Integrate RET build SRAMs to fix sleep/blfloat isolation bug in standard SRAMs.
Stepping	APL Ax
Workaround	Disable sleep/blfloat through writing PTDC register bit 30/31 (SSLPD/SBLFD – SRAM Sleep disable, SRAM Bitline float disable) using IA FW after exiting cold boot.
Impact of Workaround	Disable sleep/blfloat feature.
Notes	Audio

Issue	TDO pin driving low when inactive, sinks 20+ mA
Stepping	APL Ax
Workaround	To disable the TDO driver
Impact of Workaround	
Notes	Same WA on BYT

Issue	SPI PCI Config registers will fail Windows* System HCK PCI compliance test
Stepping	APL Ax
Workaround	1) Write B0:D13:F2 offset 0xDC (BIOS Control) bit [9] = 1'b1 -- OS function hide 2) Write B0:D13:F2 offset 0xDC (BIOS Control) bit [7] = 1'b1 -- BILD (BIOS Interface Lock Down); 3) Write PSF_3_AGNT_T0_SHDW_CFG_DIS_SPI_RS0_D13_F2 register (offset 0x2338) to set CfgDis (bit[0]) = 1'b1 for B0:D13:F2;
Impact of Workaround	
Notes	SPI, not in RDL yet and not in A0; will be implemented in E0

§